Elementos de programação em C

Compilação e visão preliminar dos programas

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Processo de compilação

O processo de compilação converte um código-fonte em código executável.

Código-fonte

Código-objeto

Código executável

Processo de compilação

O processo de compilação converte um código-fonte em código executável.

Código-fonte

Código escrito em uma linguagem de programação.

Código-objeto

Código executável

Processo de compilação

O processo de compilação converte um código-fonte em código executável.

Código-fonte

Código escrito em uma linguagem de programação.

Código-objeto

Código gerado na linguagem de máquina da arquitetura alvo.

Código executável

Processo de compilação

O processo de compilação converte um código-fonte em código executável.

Código-fonte

Código escrito em uma linguagem de programação.

Código-objeto

Código gerado na linguagem de máquina da arquitetura alvo.

Código executável

Código gerado na linguagem de máquina da arquitetura alvo, que pode ser diretamente executado pelo processador.

Etapas da compilação

O compilador C realiza a compilação dos programas-fonte em quatro etapas:

- Pré-processamento. O texto do programa é transformado lexicamente. Resultado → *unidade de compilação*.

Etapas da compilação

O compilador C realiza a compilação dos programas-fonte em quatro etapas:

- Pré-processamento. O texto do programa é transformado lexicamente. Resultado → *unidade de compilação*.
- Compilação. Análise sintática e semântica. Resultado → código assembler correspondente.

Etapas da compilação

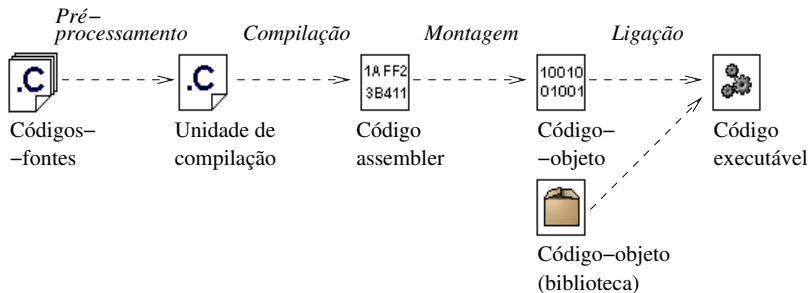
O compilador C realiza a compilação dos programas-fonte em quatro etapas:

- Pré-processamento. O texto do programa é transformado lexicamente.
  Resultado $\rightarrow$ *unidade de compilação*.
- Compilação. Análise sintática e semântica.
  Resultado $\rightarrow$ código assembler correspondente.
- Montagem. Transformação para linguagem de máquina.
  Resultado $\rightarrow$ código-objeto.

Etapas da compilação

O compilador C realiza a compilação dos programas-fonte em quatro etapas:

- Pré-processamento. O texto do programa é transformado lexicamente.
  Resultado → *unidade de compilação*.
- Compilação. Análise sintática e semântica.
  Resultado → código assembler correspondente.
- Montagem. Transformação para linguagem de máquina.
  Resultado → código-objeto.
- Ligação. Combinação dos códigos-objeto que compõem o programa.
  Resultado → código executável.

Etapas da compilação

Tipos de arquivos

- `.c` Programas-fonte: `prog.c`, `calculo.c`.
- `.h` Arquivos-cabeçalho: `prog.h`, `calculo.h`.
- `.s` Programas assembler: `prog.s`, `calculo.s`.
- `.o` Programas-objeto: `prog.o`, `calculo.o`.

Os programas executáveis não possuem uma extensão padrão. (o compilador gcc usa o nome `a.out` por omissão).

Tipos de arquivos

Arquivos-cabeçalhos são códigos-fonte que contêm a declaração de variáveis, macros e funções.

Bibliotecas são arquivos especiais que contêm o código-objeto de funções.

Tipos de arquivos

Arquivos-cabeçalhos são códigos-fonte que contêm a declaração de variáveis, macros e funções.

Bibliotecas são arquivos especiais que contêm o código-objeto de funções.

Alguns arquivos-cabeçalho e bibliotecas do sistema:

`stdio.h`	Funções de entrada e saída	`libc.a`
`math.h`	Funções matemáticas	`libm.a`

Inclusão de arquivos-cabeçalho

A inserção do código dos arquivos-cabeçalho nos programas-fonte se faz com a diretiva `#include`

- `#include <stdio.h>` inclui o arquivo-cabeçalho do sistema `stdio.h`.
- `#include "calcula.h"` inclui o arquivo-cabeçalho do usuário `calcula.h`, cujo código deve estar implementado em algum arquivo objeto, provavelmente o `calcula.o`.

Comando de compilação (gcc)

```
gcc prog.c
```

Comando de compilação (gcc)

gcc prog.c

Compila o programa que está no arquivo `prog.c` e gera um executável, armazenando-o no arquivo `a.out`.

Comando de compilação (gcc)

```
gcc prog.c
```

Compila o programa que está no arquivo `prog.c` e gera um executável, armazenando-o no arquivo `a.out`.

```
gcc prog.c aux.c ent_sai.c
```

Comando de compilação (gcc)

`gcc prog.c`

Compila o programa que está no arquivo `prog.c` e gera um executável, armazenando-o no arquivo `a.out`.

`gcc prog.c aux.c ent_sai.c`

Compila o programa cujo código está distribuído nos arquivos `prog.c`, `aux.c` e `ent_sai.c` e gera um executável no arquivo `a.out`.

Comando de compilação (gcc)

```
gcc -o prog prog.c
```

Comando de compilação (gcc)

```
gcc -o prog prog.c
```

Compila o programa que está no arquivo `prog.c` e gera um executável no arquivo `prog`.

Comando de compilação (gcc)

`gcc -o prog prog.c`

Compila o programa que está no arquivo `prog.c` e gera um executável no arquivo `prog`.

`gcc prog.c aux.c ent_sai.c -o prg_exem`

Comando de compilação (gcc)

`gcc -o prog prog.c`

Compila o programa que está no arquivo `prog.c` e gera um executável no arquivo `prog`.

`gcc prog.c aux.c ent_sai.c -o prg_exem`

Compila o programa cujo código está distribuído nos arquivos `prog.c`, `aux.c` e `ent_sai.c` e gera um executável no arquivo `prg_exem`.

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

```
gcc prog.s
```

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

`gcc prog.s`

Realiza as seguintes etapas de compilação:

- `prog.s`: montagem e ligação.

O executável é armazenando no arquivo `a.out`.

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

```
gcc prog.c aux.o ent_sai.s
```

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

`gcc prog.c aux.o ent_sai.s`

Realiza as seguintes etapas de compilação:

- `prog.c`: todas as etapas.
- `ent_sai.s`: montagem e ligação.
- `aux.o`: ligação.

O executável é armazenado no arquivo `a.out`.

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

```
gcc -o prog prog.o
```

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

`gcc -o prog prog.o`

Realiza as seguintes etapas de compilação:

- `prog.o`: ligação.

O executável é armazenado no arquivo `prog`.

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

```
gcc prog.c -o prg_exem aux.o ent_sai.s
```

Comando de compilação (gcc)

O processo de compilação preserva as etapas já realizadas:

`gcc prog.c -o prg_exem aux.o ent_sai.s`

Realiza as seguintes etapas de compilação:

- `prog.c`: todas.
- `ent_sai.s`: montagem e ligação.
- `aux.o`: ligação.

O executável é armazenado no arquivo `prg_exem`.

Compilações parciais

As opções de compilação `-E`, `-S` e `-c` permitem a realização de compilações parciais:

`-E`

Realiza apenas a etapa de pré-processamento.
Resultado: unidade de compilação.
Mostrado no terminal.

Compilações parciais

As opções de compilação `-E`, `-S` e `-c` permitem a realização de compilações parciais:

-S

Realiza as etapas de pré-processamento e compilação.
Resultado: código assembler.
Armazenado em arquivo com extensão `.s`.

Compilações parciais

As opções de compilação `-E`, `-S` e `-c` permitem a realização de compilações parciais:

`-c`

Realiza as etapas de pré-processamento, compilação e montagem.
Resultado: código-objeto.
Armazenado em arquivo com extensão `.o`.

Compilações parciais

```
gcc -E primeiro.c -o primeiro.e
```

```
#include <stdio.h>
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

Compilações parciais

```
gcc -E primeiro.c -o primeiro.e
```

```
#include <stdio.h>
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

⇒

```
extern int printf
(__const char *__restrict
__format, ...);
extern void funlockfile
(FILE *__stream)
__attribute__
 ((__nothrow__));
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

Compilações parciais

```
gcc -S primeiro.c
```

```
#include <stdio.h>
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

Compilações parciais

gcc -S primeiro.c

```
#include <stdio.h>
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

$\Rightarrow$

```
pushl %ecx
subl $4, %esp
movl $.LC0, (%esp)
call printf movl $0, %eax
addl $4, %esp
popl %ecx
popl %ebp
leal -4(%ecx), %esp
ret
```

Compilações parciais

gcc -c primeiro.c

```
#include <stdio.h>
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

Compilações parciais

gcc -c primeiro.c

```
#include <stdio.h>
int main(void) {
  printf("primeiro prog");
  return 0;
}
```

⇒

```
08 00 8D 4C 24 04 83 E4 F0 FF
71 FC 55 89 E5 51 83 EC 04 C7
04 24 00 00 00 00 E8 FC FF FF
FF B8 00 00 00 00 83 C4 04 59
5D 8D 61 FC C3 00 70 72 69 6D
65 69 72 6F 20 70 72 6F 67 72
61 6D 61 00 00 47 43 43 3A 20
```

Definindo o padrão e os avisos

```
gcc prog.c -o prg_exem aux.o ent_sai.s -std=c99 -Wall -pedantic
```

- `-std=c99` Especifica o padrão ISO/IEC 9899:1999.
- `-Wall` Força a exibição de todos os avisos do compilador.
- `-pedantic` Fornece os diagnósticos especificados pelo padrão.

Estrutura das funções C

$$\langle \mathit{Função} \rangle \quad ::= \quad \langle \mathit{TipoValorRetorno} \rangle \; \langle \mathit{NomeFunção} \rangle \; \mathbf{(} \; \langle \mathit{ListaParâmetros} \rangle \; \mathbf{)} \; \langle \mathit{CorpoFunção} \rangle$$

Estrutura das funções C

$\langle \textit{Função} \rangle \quad ::= \quad \langle \textit{TipoValorRetorno} \rangle \; \langle \textit{NomeFunção} \rangle \; \textbf{(} \; \langle \textit{ListaParâmetros} \rangle \; \textbf{)} \; \langle \textit{CorpoFunção} \rangle$

Função principal

```
int main(void) {
  return 0;
}
```

Primeiro programa

Programa-fonte prog.c

```
#include <stdio.h>
int main(void) {
  prinf("primeiro prog");
  return 0;
}
```

Primeiro programa

Programa-fonte prog.c

```
#include <stdio.h>
int main(void) {
  prinf("primeiro prog");
  return 0;
}
```

Compilação e execução

```
>gcc -o prog prog.c
>./prog
>primeiro prog
```

Desenvolvendo os primeiros programas

Importante:

- Os comandos e estruturas mostrados a seguir têm o objetivo de permitir a compreensão dos programas iniciais.
- Devem ser usados exatamente como indicado.
- Todos os comandos e estruturas serão vistos com mais detalhes no momento oportuno.

Comentários

- /* */ Compreende todas as linhas entre /* e */.
- // Compreende todos os caracteres de // até o fim da linha.

Comentários

- /* */ Compreende todas as linhas entre /* e */.
- // Compreende todos os caracteres de // até o fim da linha.

```
/* Programa exemplo */
#include <stdio.h>
int main(void) {
  /* Impressao da mensagem
   * usada como argumento
   */
  printf("primeiro programa");
  return 0;    // Comando de retorno
               // Linha em branco
}
```

Declaração de variáveis

- `int taxa;` Declara a variável `taxa`
  - do tipo `int`: pode armazenar valores inteiros.
  - não possui valor inicial.
- `int cod = 23;` Declara a variável `cod`
  - do tipo `int`: pode armazenar valores inteiros.
  - possui valor inicial 23.
- `double acel = 9.8;` Declara a variável `acel`
  - do tipo `double`: pode armazenar valores reais.
  - possui valor inicial 9,8.

Declaração de variáveis

```
int main(void) {
  int taxa, matricula;
  int qtd = 3, seq, aux;
  double salario;
  double juros, amortizacao = 2.7;
  return 0;
}
```

Atribuição de valores

- `x = 2;` Atribui o valor 2 a `x`.
- `y = 23 + x;` Atribui o valor 25 a `y`

  25 é o resultado de somar 23 e 2 (o valor de `x`).

Atribuição de valores

```
#include <stdio.h>
int main(void) {
  int x, y = 3;
  double r = 12.3, s;
  x = 2;
  y = x + 32;
  s = x + r;
  return 0;
}
```

Lendo valores do teclado

- `scanf("%d", &x);` Lê um valor inteiro do teclado armazenando-o na variável `x`.
- `scanf("%lf", &y);` Lê um valor real do teclado armazenando-o na variável `y`.

Lendo valores do teclado

```
#include <stdio.h>
int main(void) {
  int qtd = 2, taxa;
  double salario;
  scanf("%d", &qtd);
  scanf("%d", &taxa);
  scanf("%lf", &salario);
  scanf("%lf", &salario);
  return 0;
}
```

Imprimindo mensagens e valores

- `printf("Exemplo");`
  Imprime os caracteres entre aspas.
- `printf("Exemplo %d", x);`
  Imprime os caracteres entre aspas, substituindo a diretiva `%d` pelo conteúdo da variável `x`.
- `printf("Exemplo %d %f", x, y);`
  Imprime os caracteres entre aspas, substituindo a diretiva `%d` pelo conteúdo da variável `x` e a diretiva `%f` pelo conteúdo da variável `y`.

Imprimindo mensagens e valores

- `printf("valor = %d", val);`
- `printf("valor = %d e taxa = %d", val, tx);`
- `printf("Salario entre %f (menor) e %f (maior).  Taxa = %d", sal, 12 * sal, taxa);`

Imprimindo mensagens e valores

- `printf("valor = %d", val);`

```
printf("valor = %d", val);
```

- `printf("valor = %d e taxa = %d", val, tx);`
- `printf("Salario entre %f (menor) e %f (maior). Taxa = %d", sal, 12 * sal, taxa);`

Imprimindo mensagens e valores

- `printf("valor = %d", val);`
- `printf("valor = %d e taxa = %d", val, tx);`

```
printf("valor = %d e taxa = %d", val, tx);
```

- `printf("Salario entre %f (menor) e %f (maior). Taxa = %d", sal, 12 * sal, taxa);`

Imprimindo mensagens e valores

- `printf("valor = %d", val);`
- `printf("valor = %d e taxa = %d", val, tx);`
- `printf("Salario entre %f (menor) e %f (maior). Taxa = %d", sal, 12 * sal, taxa);`

```
printf("Salario entre %f (menor) e %f (maior). Taxa = %d", sal, 12 * sal, taxa);
```

Imprimindo mensagens e valores

Exercício. Elabore um programa que leia dois valores do teclado e imprima a soma dos valores lidos.

Imprimindo mensagens e valores

Exercício. Elabore um programa que leia dois valores do teclado e imprima a soma dos valores lidos.

```
#include <stdio.h>
int main(void) {
  int x, y;
  printf("Primeiro valor inteiro: ");
  scanf("%d", &x);
  printf("Segundo valor inteiro: ");
  scanf("%d", &y);
  printf("Soma = %d", x + y);
  return 0;
}
```

Desviando o fluxo de execução

```
if ( condição ) {
    comandos do bloco-então
}
```
⟶ Próximo comando após o `if`.

Se a condição for verdadeira executa os comandos do bloco então; se for falsa, prossegue após o `if`.

Desviando o fluxo de execução

```
if ( condição ) {
    comandos do bloco-então
} else {
    comandos do bloco-senão
}
```
⟶ Próximo comando após o `if`.

Se a condição for verdadeira executa os comandos do bloco então; se for falsa executa os comandos do bloco senão.

Desviando o fluxo de execução

Exercício. Elabore um programa que leia dois valores inteiros do teclado e imprima o maior valor lido.

Desviando o fluxo de execução

Exercício. Elabore um programa que leia dois valores inteiros do teclado e imprima o maior valor lido.

```
#include <stdio.h>
int main(void) {
  int x, y;
  scanf("%d", &x);
  scanf("%d", &y);
  printf("O maior numero digitado foi ");
  if (x > y) {
    printf("%d", x);
  } else {
    printf("%d", y);
  }
  return 0;
}
```

Desviando o fluxo de execução

Exercício. Elabore um programa que leia dois valores reais do teclado, a e b, e imprima a raiz da equação $ax + b = 0$.

Desviando o fluxo de execução

Exercício. Elabore um programa que leia dois valores reais do teclado, a e b, e imprima a raiz da equação $ax + b = 0$.

```
#include <stdio.h>
int main(void) {
  double a, b;
  scanf("%lf", &a);
  scanf("%lf", &b);
  printf("Equacao: %fx + %f = 0\n", a, b);
  if (a == 0.0) {
    printf("Nao existe raiz\n");
  } else {
    printf("Raiz = %f \n", (-b / a));
  }
  return 0;
}
```

Chamando funções

- As funções são chamadas pelo nome com que são declaradas.
- As funções devem ser declaradas antes de serem usadas.
- Se a funções possui parâmetros, seus valores devem ser fornecidos por ocasião da chamada.
- Após a execução da função o fluxo retorna para o ponto seguinte ao da chamada.
- Se a função retorna valor, o valor retornado pode ser armazenado ou impresso.

Chamando funções

Algumas funções predefinidas:

`scanf`	Lê valores do teclado.
`printf`	Imprime valores no monitor de vídeo.
`pow(a,b)`	Retorna o valor a^b.
`sqrt(a)`	Retorna o valor $\sqrt{a}$.

Chamando funções

Exercício. Elabore um programa que leia um valor real do teclado e imprima a raiz quadrada e, em seguida, o cubo do valor lido.

Chamando funções

Exercício. Elabore um programa que leia um valor real do teclado e imprima a raiz quadrada e, em seguida, o cubo do valor lido.

```
#include <stdio.h>
#include <math.h>
int main(void) {
  double x, r, p;
  scanf("%lf", &x);
  r = sqrt(x);
  p = pow(x, 3);
  printf("Raiz de %f= %f\n", x, r);
  printf("Cubo de %f = %f\n", x, p);
  return 0;
}
```

Obsrvação. Para funcionar este programa deve ser compilado com a opção `-lm`*, que indica o uso da biblioteca matemática.*

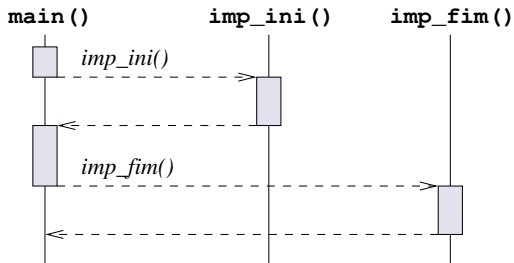
Declarando funções

Várias funções podem ser declaradas/definidas em um mesmo arquivo.

```
#include <stdio.h>
void imp_ini(void) {
   printf("impressao do cabecalho\n");
}
void imp_fim(void) {
   printf("fim de programa\n");
}

int main (void) {
   int x;
   imp_ini();
   scanf("%d", &x);
   printf("dobro = %d\n", (2 * x));
   imp_fim();
   return 0;
}
```

O programa ao lado define as funções `imp_ini` e `imp_fim`. Ambas sem parâmetro e sem valor de retorno.

Fluxo de execução

Fluxo de execução na chamada a uma função.

Variáveis globais

- As variáveis globais são declaradas fora do corpo das funções do programa.
- O escopo de uma variável global vai do ponto da declaração até o fim da unidade de compilação onde são declaradas.
- Qualquer função no escopo de uma variável global pode utilizá-la.

Variáveis globais

```
#include <stdio.h>
int x;
void lerdados(void) {
  printf("Digite um valor inteiro: ");
  scanf("%d", &x);
}
void impdados(void) {
  printf("dobro = %d\n", (2 * x));
}
int main(void) {
  lerdados();
  impdados();
  printf("fim programa\n");
  return 0;
}
```

Unidade de compilação

Os programas podem ser codificados em um ou vários arquivos-fonte e podem ser distribuídos em varias unidades de compilação, compiladas separadamente.

Observação: os novatos em programação podem desconsiderar os eslaides restantes desta apresentação, retomando-os posteriormente, após terem avançado no entendimento da linguagem C.

Unidade de compilação

Exemplo base

Codificar em um único arquivo um programa que leia do teclado três números reais, a, b e c, e imprime as raízes reais da equação quadrática $ax^2 + bx + c = 0$.

```c
#include <stdio.h>
#include <math.h>
int main(void) {
  double a, b, c, delta, r1,r2;
  printf("Digite o coeficiente a: ");
  scanf("%lf", &a);
  printf("Digite o coeficiente b: ");
  scanf("%lf", &b);
  printf("Digite o coeficiente c: ");
  scanf("%lf", &c);
  delta = pow(b, 2) - 4 * a * c;
  if (delta >= 0.0) {
    r1 = (-b + sqrt(delta)) / (2 * a);
    r2 = (-b - sqrt(delta)) / (2 * a);
  }
  if (delta >= 0.0) {
    printf("Raiz r1 = %f\n", r1);
    printf("Raiz r2 = %f\n", r2);
  } else {
    printf("Sem raizes reais\n");
  }
  return 0;
}
```

Unidade de compilação — múltiplos arquivos-fonte

O programa do exemplo base será codificado em três unidades de compilação:

1. `ent_sai.c`, contendo as funções de leitura dos números reais e impressão das raízes da equação.
2. `calcula.c`, contendo as funções para o cálculo do delta e das raízes.
3. `eq_quad2.c`, contendo a função `main`, que chama as demais.

Unidade de compilação — múltiplos arquivos-fonte

eq_quad.c

```
#include <stdio.h>
#include <math.h>
int main(void) {
   double a, b, c, delta, r1, r2;
   printf("Digite o coeficiente a: ");
   scanf("%lf", &a);
   printf("Digite o coeficiente b: ");
   scanf("%lf", &b);
   printf("Digite o coeficiente c: ");
   scanf("%lf", &c);
   delta = pow(b, 2) - 4 * a * c;
   if (delta >= 0) {
      r1 = (-b + sqrt(delta)) / (2 * a);
      r2 = (-b - sqrt(delta)) / (2 * a);
   }
   if (delta >= 0) {
      printf("Raiz r1 = %f\n", r1);
      printf("Raiz r2 = %f\n", r2);
   } else {
      printf("Sem raizes reais\n");
   }
   return 0;
}
```

ent_sai.c

```
ler_dados()
imp_resultado()
```

calcula.c

```
calcula_delta()
calcula_raizes()
```

eq_quad2.c

```
int main(void) {
   ler_dados()
   calcula_raizes()
   imp_resultado()
}
```

Unidade de compilação — múltiplos arquivos-fonte

ent_sai.c

```
#include <stdio.h>
double a, b, c, delta, r1, r2;
void ler_dados(void) {
  printf("Digite o coeficiente a: ");
  scanf("%lf", &a);
  printf("Digite o coeficiente b: ");
  scanf("%lf", &b);
  printf("Digite o coeficiente c: ");
  scanf("%lf", &c);
}
void imp_resultado(void) {
  if (delta >= 0.0) {
    printf("Raiz r1 = %f\n", r1);
    printf("Raiz r2 = %f\n", r2);
  } else {
    printf("Sem raizes reais\n");
  }
}
```

Unidade de compilação — múltiplos arquivos-fonte

calcula.c

```
#include <math.h>
extern double a, b, c, delta, r1, r2;
void calcula_delta(void) {
  delta = pow(b, 2.0) - 4.0 * a * c;
}
void calcula_raizes(void) {
  calcula_delta();
  if (delta >= 0.0) {
     r1 = (-b + sqrt(delta)) / (2.0 * a);
     r2 = (-b - sqrt(delta)) / (2.0 * a);
  }
}
```

Unidade de compilação — múltiplos arquivos-fonte

eq_quad2.c

```
extern void ler_dados(void);
extern void calcula_raizes(void);
extern void imp_resultado(void);
int main(void) {
  ler_dados();
  calcula_raizes();
  imp_resultado();
  return 0;
}
```

Unidade de compilação — múltiplos arquivos-fonte

eq_quad2.c

```
extern void ler_dados(void);
extern void calcula_raizes(void);
extern void imp_resultado(void);
int main(void) {
  ler_dados();
  calcula_raizes();
  imp_resultado();
  return 0;
}
```

Para ser corretamente referenciadas as funções `ler_dados`, `calcula_raizes` e `imp_resultado` devem ser previamente declaradas.

Unidade de compilação — múltiplos arquivos-fonte

eq_quad2.c

```
extern void ler_dados(void);
extern void calcula_raizes(void);
extern void imp_resultado(void);
int main(void) {
  ler_dados();
  calcula_raizes();
  imp_resultado();
  return 0;
}
```

Para ser corretamente referenciadas as funções `ler_dados`, `calcula_raizes` e `imp_resultado` devem ser previamente declaradas.

Usa-se a palavra-chave `extern` na declaração de uma função para indicar que pode estar implementada em outra unidade de compilação.

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`:

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em ?

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`:

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em ?

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em `calcula.o`.

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em `calcula.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `eq_quad2.c`:

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em `calcula.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `eq_quad2.c`: `gcc -c eq_quad2.c`
  - Resultado é armazenado em ?

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em `calcula.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `eq_quad2.c`: `gcc -c eq_quad2.c`
  - Resultado é armazenado em `eq_quad2.o`.

Compilando múltiplas unidades

- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em `calcula.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `eq_quad2.c`: `gcc -c eq_quad2.c`
  - Resultado é armazenado em `eq_quad2.o`.

Se os objetos estão disponíveis, o executável pode ser gerado do seguinte modo:

```
gcc -o eq_quad2 eq_quad2.o ent_sai.o calcula.o
```

Compilando múltiplas unidades

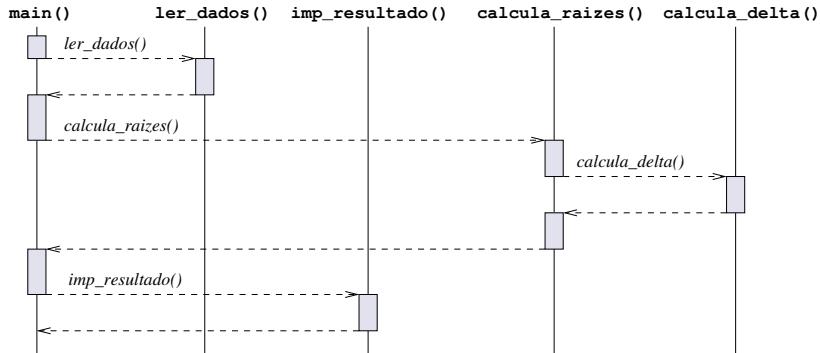
- Comando para gerar o código-objeto que implementa as funções do arquivo `ent_sai.c`: `gcc -c ent_sai.c`
  - Resultado é armazenado em `ent_sai.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `calcula.c`: `gcc -c calcula.c`
  - Resultado é armazenado em `calcula.o`.
- Comando para gerar o código-objeto que implementa as funções do arquivo `eq_quad2.c`: `gcc -c eq_quad2.c`
  - Resultado é armazenado em `eq_quad2.o`.

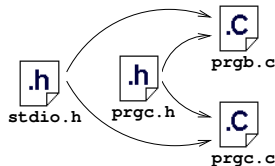
Também é possível gerar o executável com um único comando:

```
gcc -o eq_quad2 eq_quad2.c ent_sai.c calcula.c
```

Unidade de compilação — Fluxo de execução

O programa executável contém todas as funções implementadas, independentemente de terem sido codificadas em um único arquivo ou em arquivos separados.

O fluxo de execução do exemplo base permanece o mesmo em ambas as versões (única ou múltiplas unidades de compilação):

Arquivos-cabeçalho

Os arquivos-cabeçalho contém declarações que são codificadas uma única vez e podem ser usadas várias vezes, sempre que um programa precisa fazer referência às funções declaradas.

Arquivos-cabeçalho

Todo arquivo contendo funções e variáveis referidas a partir de outros arquivos deve ser codificado em duas partes:

1. Um arquivo-cabeçalho, com as declarações das funções e variáveis que podem ser referidas em outros arquivos, e
2. Um arquivo-implementação, com o código que implementa essas funções e variáveis.

Arquivos-cabeçalho — exemplo base

ent_sai.h

```
extern void ler_dados(void);
extern void imp_resultado(void);
extern double a, b, c, delta, r1, r2;
```

ent_sai.c

```
#include <stdio.h>
#include "ent_sai.h"
double a, b, c, delta, r1, r2;
void ler_dados(void) {
  printf("Digite o coeficiente a: ");
  scanf("%lf", &a);
  printf("Digite o coeficiente b: ");
  scanf("%lf", &b);
  printf("Digite o coeficiente c: ");
  scanf("%lf", &c);
}
```

continua...

Arquivos-cabeçalho — exemplo base

ent_sai.c ...continuação

```
void imp_resultado(void) {
  if (delta >= 0.0) {
    printf("Raiz r1 = %f\n", r1);
    printf("Raiz r2 = %f\n", r2);
  } else {
    printf("Sem raizes reais\n");
  }
}
```

Arquivos-cabeçalho — exemplo base

calcula.h

```
extern void calcula_raizes(void);
```

calcula.c

```
#include <math.h>
#include "calcula.h"
#include "ent_sai.h"
void calcula_delta(void) {
  delta = pow(b, 2.0) - 4.0 * a * c;
}
void calcula_raizes(void) {
  calcula_delta();
  if (delta >= 0.0) {
     r1 = (-b + sqrt(delta)) / (2.0 * a);
     r2 = (-b - sqrt(delta)) / (2.0 * a);
  }
}
```

Arquivos-cabeçalho — exemplo base

eq_quad2.c

```
#include "ent_sai.h"
#include "calcula.h"
int main(void) {
  ler_dados();
  calcula_raizes();
  imp_resultado();
  return 0;
}
```

Arquivos-cabeçalho — exemplo base

eq_quad2.c

```
#include "ent_sai.h"
#include "calcula.h"
int main(void) {
  ler_dados();
  calcula_raizes();
  imp_resultado();
  return 0;
}
```

Com a possível exceção do arquivo que contém a função main, cada arquivo deve incluir

- seu próprio arquivo-cabeçalho,
- além dos arquivos-cabeçalho que declaram as funções e variáveis que eles referenciam.

Modularização

A modularização (com o uso de múltiplas unidades de compilação) é necessária no desenvolvimento de grandes programas, pois facilita

- a elaboração da solução,
- a realização dos testes e
- a manutenção do código gerado.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Tipos de dados

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Tipos de dados

Um tipo de dado caracteriza um conjunto de valores, determinando:

Natureza. Caracteriza o tipo representado, que pode ser, por exemplo, um caractere, um número inteiro, um número real ou uma cadeia de caracteres.

Tamanho. Determina o tamanho em bits necessário para armazenar os valores do tipo.

Representação. Determina a forma como os bits armazenados devem ser interpretados.

Imagem ou faixa de representação. Determina a faixa de valores válidos para o tipo.

A expressão usada para identificar um tipo de dado é chamada de *especificador de tipo*.

Tipos de dados

Exemplo

Se o tipo de dado `tipo_exem` é usado para caracterizar os números inteiros de 8 bits armazenados na forma de complemento-2, temos:

Especificador:	tipo_exem.
Natureza:	números inteiros.
Tamanho:	8 bits.
Imagem:	$[-128, 127]$.
Representação:	complemento-2.

Os inteiros 32 e -104 são representáveis no tipo `tipo_exem`. Já o número real $32,0$ não é representável nesse tipo, embora possa ser convertido em um valor representável: o inteiro 32.

Tipos básicos da linguagem C

Classificação		Tipos básicos
		`char`

Inteiros sinalizados	Inteiros sinalizados padrões	`signed char`
		`short int`
		`int`
		`long int`
		`long long int`

	Inteiros sinalizados estendidos	

Inteiros não sinalizados	Inteiros não sinalizados padrões	`unsigned char`
		`unsigned short int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`
		`_Bool`

	Inteiros não sinalizados estendidos	

Ponto flutuante	Reais de ponto flutuante	`float`
		`double`
		`long double`

	Complexos	`float _Complex`
		`double _Complex`
		`long double _Complex`

Classificações alternativas dos tipos de dados

<table>
<tr><td>Caracteres</td><td colspan="2"><code>char</code>
<code>signed char</code>
<code>unsigned char</code></td></tr>
<tr><td>Inteiros</td><td colspan="2"><code>char</code>
Inteiros sinalizados
Inteiros não sinalizados
Tipos enumerados</td></tr>
<tr><td>Reais</td><td colspan="2">Inteiros
Reais de ponto flutuante</td></tr>
<tr><td rowspan="3">Aritméticos</td><td colspan="2">Inteiros</td></tr>
<tr><td rowspan="2">Ponto flutuante</td><td>Reais de ponto flutuante</td></tr>
<tr><td>Complexos</td></tr>
<tr><td>Escalares</td><td colspan="2">Aritméticos
Ponteiros</td></tr>
<tr><td>Agregados</td><td colspan="2">Estruturas
Vetores</td></tr>
<tr><td>Domínio real</td><td colspan="2">Inteiros e Reais de ponto flutuante</td></tr>
<tr><td>Domínio complexo</td><td colspan="2">Complexos</td></tr>
</table>

Tipos caracteres

- Cada caractere é representado por um código numérico.
- Os valores podem ser sinalizados ou não.
- Tamanho do tipo é dado pela macro CHAR_BIT.

Tipo caractere	Valor mínimo	Valor máximo
char	CHAR_MIN	CHAR_MAX
signed char	SCHAR_MIN	SCHAR_MAX
unsigned char	0	UCHAR_MAX

As macros do tamanho e valores mínimo e máximo estão declaradas em limits.h.

Conjunto básico de caracteres

- Conjunto básico de caracteres-fonte
- Conjunto básico de caracteres de execução

Conjunto básico de caracteres

- Conjunto básico de caracteres-fonte
- Conjunto básico de caracteres de execução

Padrão ASCII

0	nul	16	dle	32		48	0	64	@	80	P	96	`	112	p
1	sch	17	dc1	33	!	49	1	65	A	81	Q	97	a	113	q
2	stx	18	dc2	34	"	50	2	66	B	82	R	98	b	114	r
3	etx	19	dc3	35	#	51	3	67	C	83	S	99	c	115	s
4	eot	20	dc4	36	$	52	4	68	D	84	T	100	d	116	t
5	enq	21	nak	37	%	53	5	69	E	85	U	101	e	117	u
6	ack	22	syn	38	&	54	6	70	F	86	V	102	f	118	v
7	bel	23	etb	39	'	55	7	71	G	87	W	103	g	119	w
8	bs	24	can	40	(	56	8	72	H	88	X	104	h	120	x
9	ht	25	em	41	)	57	9	73	I	89	Y	105	i	121	y
10	lf	26	sub	42	*	58	:	74	J	90	Z	106	j	122	z
11	vt	27	esc	43	+	59	;	75	K	91	[	107	k	123	{
12	ff	28	fs	44	,	60	<	76	L	92	\	108	l	124	|
13	cr	29	gs	45	-	61	=	77	M	93	]	109	m	125	}
14	so	30	rs	46	.	62	>	78	N	94	^	110	n	126	~
15	si	31	us	47	/	63	?	79	O	95	_	111	o	127	del

Caracteres multibytes

- Cada caractere é representado por uma sequência de bytes (caracteres).
- Requer a noção de estado para indicar como cada sequência de caracteres é representada.

Caracteres multibytes

Exemplo

O padrão JIS X 208 define uma codificação em que cada caractere japonês é representado por um par de caracteres do conjunto básico de caracteres. Neste padrão o caractere 殲 é representado pelos caracteres `p` e `6`.

1. A sequência `<ESC>`, `$` e `B` inicia o estado em que os caracteres são interpretados segundo o padrão JIS X 208.
2. A sequência `<ESC>`, `(` e `B` termina o estado JIS X 208.

Desse modo, a sequência

```
p  6  p  <ESC>  $  B  p  6  <ESC>  (  B  p  6  n
```

representa, de fato, uma sequência de 7 caracteres.

Caracteres multibytes

Exemplo

O padrão JIS X 208 define uma codificação em que cada caractere japonês é representado por um par de caracteres do conjunto básico de caracteres. Neste padrão o caractere [illegible] é representado pelos caracteres p e 6.

1. A sequência <ESC>, $ e B inicia o estado em que os caracteres são interpretados segundo o padrão JIS X 208.
2. A sequência <ESC>, (e B termina o estado JIS X 208.

Desse modo, a sequência

p	6	p	<ESC>	$	B	p	6	<ESC>	(	B	p	6	n
p	6	p				[illegible]					p	6	n

representa, de fato, uma sequência de 7 caracteres.

Padrão Unicode

- Capaz de representar os caracteres de todos os alfabetos conhecidos.
- Existem vários estilos de codificação:
  - `UTF-32` utiliza 4 bytes (octetos) para representar os caracteres.
  - `UTF-16` utiliza 2 bytes para representar os caracteres, podendo também representá-los como pares de 16 bits.
  - `UTF-8` utiliza de 1 a 4 bytes para representar os caracteres.
- Não usa sequência de transição. O estado em que cada caractere é interpretado é definido pelo próprio caractere:

`0xxxxxxx`	1 octeto
`110xxxxx 10xxxxxx`	2 octetos
`1110xxxx 10xxxxxx 10xxxxxx`	3 octetos
`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	4 octetos

Caracteres estendidos

- São representados por uma mesma quantidade de bytes.
- Costuma-se usar
  - uma codificação Unicode de tamanho fixo (UTF-32)
  - uma adaptação de uma codificação Unicode de tamanho variável (UTF-8)
  - O padrão UCS (*Universal Character Set*, semelhante ao Unicode).

O tipo `wchar_t` (cabeçalho `wchar.h`) é usado para designar caracteres estendidos.

Os tipos `char16_t` e `char32_t` (cabeçalho `uchar.h`) designam caracteres estendidos codificados no padrão UTF-16 e UTF-32, respectivamente.

Tamanhos e valores dos tipos inteiros

Tipo inteiro	Tamanho	Valor mínimo	Valor Máximo
`signed char`	8	−128	127
`short int`	16	−32.768	32.767
`int`	32	−2.147.483.648	2.147.483.647
`long int`	32	−2.147.483.648	2.147.483.647
`long long int`	64	−9.223.372.036.854.775.808	9.223.372.036.854.775.807
`unsigned char`	8	0	255
`unsigned short int`	16	0	65535
`unsigned int`	32	0	4.294.967.295
`unsigned long int`	32	0	4.294.967.295
`unsigned long long int`	64	0	18.446.744.073.709.551.615

Valores adotados pelo compilador gcc para uma arquitetura de 32 bits e complemento-2 para representação de negativos.

Tamanhos e valores dos tipos inteiros

- Os valores mínimo e máximo para cada tipo inteiro são determinados pelas macros do arquivo-cabeçalho `limits.h`, mostradas a seguir:

Tipo inteiro	mínimo	máximo	Tipo inteiro	mínimo	máximo
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`	`unsigned char`	0	`UCHAR_MAX`
`short int`	`SHRT_MIN`	`SHRT_MAX`	`unsigned short int`	0	`USHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`	`unsigned int`	0	`UINT_MAX`
`long int`	`LONG_MIN`	`LONG_MAX`	`unsigned long int`	0	`ULONG_MAX`
`long long int`	`LLONG_MIN`	`LLONG_MAX`	`unsigned long long int`	0	`ULLONG_MAX`

Tamanhos e valores dos tipos inteiros

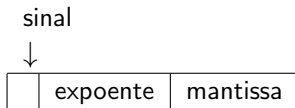
- Os valores mínimo e máximo para cada tipo inteiro são determinados pelas macros do arquivo-cabeçalho `limits.h`, mostradas a seguir:

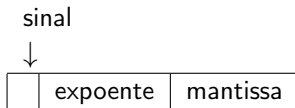
Tipo inteiro	mínimo	máximo	Tipo inteiro	mínimo	máximo
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`	`unsigned char`	0	`UCHAR_MAX`
`short int`	`SHRT_MIN`	`SHRT_MAX`	`unsigned short int`	0	`USHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`	`unsigned int`	0	`UINT_MAX`
`long int`	`LONG_MIN`	`LONG_MAX`	`unsigned long int`	0	`ULONG_MAX`
`long long int`	`LLONG_MIN`	`LLONG_MAX`	`unsigned long long int`	0	`ULLONG_MAX`

- O tamanho atribuído a cada tipo dependente da arquitetura.
- O padrão estabelece a seguinte relação de ordem entre eles:

`signed char` $\leq$ `short int` $\leq$ `int` $\leq$ `long int` $\leq$ `long long int`

Tipos reais de ponto flutuante

- São implementados com a representação sinal, mantissa e expoente:

Tipos reais de ponto flutuante

- São implementados com a representação sinal, mantissa e expoente:

- É comum a adoção do padrão IEC 60559 (IEEE 754).
- Os tamanhos usuais de cada tipo são:

Tipo	Sinal	Expoente	Mantissa
`float`	1	8	23
`double`	1	11	52
`long double`	1	64	63

Tipos reais de ponto flutuante

- Valores positivos mínimo e máximo para cada tipo (definidos no arquivo-cabeçalho `float.h`):

Tipo	Menor valor	Maior valor
`float`	`FLT_MIN`	`FLT_MAX`
`double`	`DBL_MIN`	`DBL_MAX`
`long double`	`LDBL_MIN`	`LDBL_MAX`

- Valores usuais para uma arquitetura de 32 bits:

Tipo	Tamanho	Menor valor	Maior valor
`float`	32	$1,17549 \times 10^{-38}$	$3,40282 \times 10^{+38}$
`double`	64	$2,22507 \times 10^{-308}$	$1,79769 \times 10^{+308}$
`long double`	128	$3,3621 \times 10^{-4932}$	$1,18973 \times 10^{+4932}$

Tipos reais de ponto flutuante

O cabeçalho `float.h` declara as seguintes macros que podem ser usadas para tratar imprecisão nas operações com valores de ponto flutuante:

Tipo	Macro	Valor máximo
`float`	`FLT_EPSILON`	1×10^{-5}
`double`	`DBL_EPSILON`	1×10^{-9}
`long double`	`LDBL_EPSILON`	1×10^{-9}

Cada macro $\langle \mathit{TIPO} \rangle$`_EPSILON` representa o menor valor positivo tal que $1,0 + \langle \mathit{TIPO} \rangle$`_EPSILON` $\neq 1,0$

Tipos complexos

Compostos de duas partes: real e imaginária.

Tipo complexo	Tipo real de ponto flutuante associado	Tipo das partes real e imaginária
`float _Complex`	`float`	`float`
`double _Complex`	`double`	`double`
`long double _Complex`	`long double`	`long double`

Tipos complexos

Compostos de duas partes: real e imaginária.

Tipo complexo	Tipo real de ponto flutuante associado	Tipo das partes real e imaginária
`float _Complex`	`float`	`float`
`double _Complex`	`double`	`double`
`long double _Complex`	`long double`	`long double`

O cabeçalho `complex.h` declara funções para obtenção das partes de um tipo complexo:

`creal(expr)`	retorna a parte real de `expr`, como um valor do tipo `double`.
`cimag(expr)`	retorna a parte imaginária de `expr`, como um valor do tipo `double`.

Tipos complexos

A macro I, que corresponde à constante imaginária, também é definida no arquivo-cabeçalho complex.h:

Exemplo

No trecho de programa a seguir as variáveis a, b e c são declaradas como de tipos complexos.

```
    double _Complex a = 2.4 + 0.5 * I;
    float _Complex b = 2 * a;
    long double _Complex c = 10 + 2.23
                        + 2.2 * I + 0.5 * I;
```

Tipos derivados

Tipo vetor. Representa sequências de valores de um mesmo tipo.

Tipo estrutura. Representa sequências de valores, possivelmente de diferentes tipos.

Tipo união. Representa valores sobrepostos, possivelmente de diferentes tipos.

Tipo função. Representa funções.

Tipo ponteiro. Representa valores que são endereços para objetos de tipos específicos.

Tipo estrutura

As estruturas permitem declarar valores estruturados, contendo diversos componentes.

⟨*Estrutura*⟩	::=	**struct** [⟨*Etiqueta*⟩] **{** ⟨*ListaComponentes*⟩ **}**
⟨*União*⟩	::=	**union** [⟨*Etiqueta*⟩] **{** ⟨*ListaCompnentes*⟩ **}**
⟨*ListaComponentes*⟩	::=	⟨*DeclTipo*⟩ ⟨*ListaDecl*⟩ **;**
	|	⟨*ListaComponentes*⟩ ⟨*DeclTipo*⟩ ⟨*ListaDecl*⟩ **;**
⟨*ListaDecl*⟩	::=	⟨*Declarador*⟩
	|	⟨*DeclCampoBits*⟩
	|	⟨*ListaDecl*⟩ **,** ⟨*Declarador*⟩
	|	⟨*ListaDecl*⟩ **,** ⟨*DeclCampoBits*⟩
⟨*DeclCampoBits*⟩	::=	[⟨*Declarador*⟩] **:** ⟨*QtdBits*⟩
⟨*Declarador*⟩	::=	Declarador de variáveis, vetores, funções, ponteiros, enumerações, estruturas ou uniões.

Tipo estrutura

Na especificação de uma estrutura, seus componentes não podem ser iniciados nem possuir classe de armazenamento.

Válida

```
struct {
  char cod, tp;
  int valor;
  double taxa;
}
```

Inválida

```
struct {
  char cod, tp;
  static int valor;
  double taxa = 2.3;
}
```

Tipo estrutura

Cada especificação define um tipo estrutura diferente, mesmo que as especificações sejam idênticas.

Exemplo

```
struct {
  int a;
  char b;
} aux, cod;
```

```
struct {
  int a;
  char b;
} taxa;
```

As variáveis aux e cod possuem o mesmo tipo, diferente do tipo da variável taxa.

Tipo estrutura

O uso de etiqueta permite referências posteriores a um tipo estrutura já declarado.

Exemplo

```
struct exem {
  int a;
  char b;
} aux, cod;
```

```
struct exem taxa;
```

Agora a variável `taxa` possui o mesmo tipo que as variáveis `aux` e `cod`.
A expressão `struct exem` refere-se à mesma estrutura declarada anteriormente com a etiqueta `exem`.

Tipo união

- As uniões permitem o armazenamento seletivo de valores de diversos tipos em um mesmo espaço de memória.
- Na especificação de uma união, seus componentes não podem ser iniciados nem possuir classe de armazenamento.

Válida

```
union {
  char cod, tp;
  int valor;
  double taxa;
}
```

Inválida

```
union {
  char cod, tp;
  static int valor;
  double taxa = 2.3;
}
```

Tipo união

Cada especificação define um tipo união diferente, mesmo que as especificações sejam idênticas.

Exemplo

```
union {
  int a;
  char b;
} aux, cod;
```

```
union {
  int a;
  char b;
} taxa;
```

As variáveis aux e cod possuem o mesmo tipo, diferente do tipo da variável taxa.

Tipo união

O uso de etiqueta permite referências posteriores a um tipo união já declarado.

Exemplo

```
union exem {
  int a;
  char b;
} aux, cod;
```

```
union exem taxa;
```

Agora a variável `taxa` possui o mesmo tipo que as variáveis `aux` e `cod`. A expressão `union exem` refere-se à mesma união declarada anteriormente com a etiqueta `exem`.

Campos de bits

- Os campos de bits são componentes de estruturas e uniões
- Não são tipos de dados: apenas designam um número determinado de bits interpretados como um valor d um tipo inteiro.
- A quantidade de bits de um campo de bits não pode exceder o tamanho do tipo inteiro especificado.

⟨*DeclCampoBits*⟩ ::= [⟨*QualifTipo*⟩] ⟨*Tipo*⟩ [⟨*Identificador*⟩] **:** ⟨*QtdBits*⟩

Exemplo

```
struct {
  _Bool estado:1;
  char op;
  int valA: 4;
}
```

```
struct {
  char op;
  unsigned int valB: 5;
}
```

```
union {
  char op;
  int : 5;
}
```

Tipos incompletos

Os tipos incompletos não possuem informação suficiente para determinar o espaço necessário para armazenar seus valores em memória. Ocorrem quando se especifica:

- um tipo vetor sem definição de tamanho,
- um tipo estrutura de conteúdo desconhecido,
- um tipo união de conteúdo desconhecido,
- um tipo enumerado sem declarar suas constantes, ou
- o tipo `void`.

Tipos incompletos

Os tipos incompletos não possuem informação suficiente para determinar o espaço necessário para armazenar seus valores em memória. Ocorrem quando se especifica:

- um tipo vetor sem definição de tamanho,
- um tipo estrutura de conteúdo desconhecido,
- um tipo união de conteúdo desconhecido,
- um tipo enumerado sem declarar suas constantes, ou
- o tipo `void`.

`void`

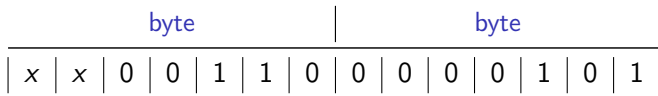
Especifica um conjunto de valores vazio. É um tipo incompleto que não pode ser completado.

Representação dos valores

Todos os tipos em C, exceto os campos de bits, são armazenados como uma sequência contígua de um ou mais bytes.

Exemplo

Em uma arquitetura em que o tamanho do byte é 7 bits, o armazenamento de valores de um tipo que ocupa 12 bits requer dois bytes, resultando em 2 bits não utilizados:

byte							byte						
x	x	0	0	1	1	0	0	0	0	0	1	0	1

O caractere x representa os bits não utilizados.

Valoração e preenchimento

- Bits de valoração. Usados para a interpretação do valor armazenado.
- Bits de preenchimento. Não utilizados

- A precisão de um tipo inteiro é o número de bits usado para representar os seus valores, excluindo os bits de sinal e preenchimento.
- O tamanho é o número de bits de valoração mais o bit de sinal.

Alinhamento dos bits

- Cada tipo possui um requisito de alinhamento que indica como os valores do tipo devem ser alocados.
- O alinhamento de cada tipo depende da implementação e é expresso como um múltiplo de byte.

Representação dos inteiros não sinalizados

- Representados por bits de valoração e preenchimento.
- O conteúdo dos bits de preenchimento não é especificado pelo padrão.

Representação dos inteiros não sinalizados

- Representados por bits de valoração e preenchimento.
- O conteúdo dos bits de preenchimento não é especificado pelo padrão.

Exemplo

Se o tamanho do byte é igual a 7 bits e o tipo `unsigned short` é definido com 8 bits, então todo valor desse tipo ocupa 14 bits.
A tabela a seguir mostra duas possíveis configurações de bits para armazenar o valor 73:

Preenchimento						Valoração							
0	0	0	0	0	0	0	1	0	0	1	0	0	1
0	1	0	1	0	1	0	1	0	0	1	0	0	1

Representação dos inteiros sinalizados

- Representados por bits de sinal, valoração e preenchimento.
- O conteúdo dos bits de preenchimento não é especificado pelo padrão.

Representação dos inteiros sinalizados

- Representados por bits de sinal, valoração e preenchimento.
- O conteúdo dos bits de preenchimento não é especificado pelo padrão.

Exemplo

Se o tamanho do byte é igual a 7 bits e o tipo `signed short` é definido com 8 bits, então todo valor desse tipo ocupa 14 bits.
A tabela a seguir mostra duas possíveis configurações de bits para armazenar o valor -84 (negativos em complemento-2):

Sinal ↓	Preenchimento						Valoração						
1	0	0	0	0	0	0	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	1	0	1	1	0	0

Representação dos caracteres

- Os tipos caracteres são sempre armazenados em 1 byte.
- Não possuem bits de preenchimento.

Representação de estruturas

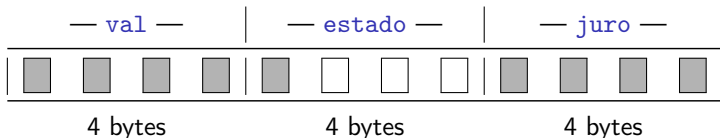
- A representação de uma estrutura corresponde à alocação ordenada dos seus componentes.
- Pode haver bits de preenchimento no interior ou no final da estrutura.

Representação de estruturas

- A representação de uma estrutura corresponde à alocação ordenada dos seus componentes.
- Pode haver bits de preenchimento no interior ou no final da estrutura.

Exemplo

Considerando os alinhamentos:
`float` = 4 bytes, `char` = 1 byte,
estrutura = 4 bytes

```
struct {
  float val;
  char  estado;
  float juro;
} aux;
```

Representação de uniões

- A memória alocada a uma união corresponde ao espaço necessário para armazenar o maior de seus componentes.
- A representação varia segundo o componente armazenado.

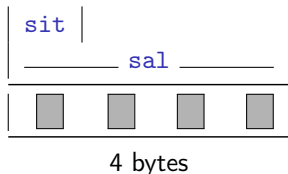
Representação de uniões

- A memória alocada a uma união corresponde ao espaço necessário para armazenar o maior de seus componentes.
- A representação varia segundo o componente armazenado.

Exemplo

Considerando os alinhamentos:
float = 4 bytes, char = 1 byte,
união = 4 bytes

```
union exem {
  char sit;
  float sal;
} aux;
```

sit
sal
4 bytes

Representação de campos de bits

- São alocados em unidades endereçáveis de memória.
- Se um campo não cabe na unidade corrente, uma nova unidade é alocada.
- Um campo de bits podem ou não cruzar as unidades de endereçamento (comportamento dependente da implementação).

Campos de bits

Exemplo

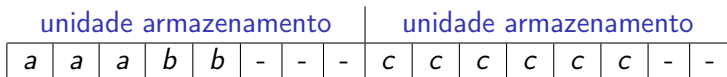
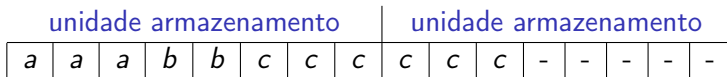
Considerando o tamanho da unidade endereçável = 8 bits.

```
struct {
  int cbA: 3;
  int cbB: 2;
  int cbC: 6;
}
```

A alocação do campo `cbC` pode ocorrer como:

unidade armazenamento								unidade armazenamento							
a	*a*	*a*	*b*	*b*	-	-	-	*c*	*c*	*c*	*c*	*c*	*c*	-	-

ou como:

unidade armazenamento								unidade armazenamento							
a	*a*	*a*	*b*	*b*	*c*	*c*	*c*	*c*	*c*	*c*	-	-	-	-	-

Conversão de tipos

- Expansiva
- Restritiva

Tipo inteiro em tipo inteiro

1. Se o valor original pode ser representado no novo tipo, então o novo valor é idêntico ao valor original.
2. Se o valor original não pode ser representado, então
   - Se o novo tipo é não sinalizado, o valor original é reduzido módulo 2^N, onde N é o tamanho do tipo alvo.
   - Se o novo tipo é sinalizado, a conversão é dependente da implementação, podendo não ser realizada e provocar um sinal de erro.

   Observação. O compilador gcc estende o procedimento de reduzir um valor módulo 2^N: o fator 2^N é adicionado ao (ou subtraído do) valor original até que este esteja na faixa de representação do tipo alvo.

Tipo inteiro em tipo inteiro

Redução módulo 2^N

- Equivale a subtrair 2^N do valor, repetidas vezes, até que o resultado esteja na faixa $[0, 2^N - 1]$.
- Se o valor original for negativo, deve-se adicionar 2^N repetidas vezes.

A redução módulo 2^N é definida quando o tipo do valor reduzido é não negativo.

Tipo inteiro em tipo inteiro

Exemplo

Considerando o tamanho do tipo `unsigned short int` $= 16$, converter o valor -73.538 do tipo `int` no tipo `unsigned short int`.

Tipo inteiro em tipo inteiro

Exemplo

Considerando o tamanho do tipo `unsigned short int` $= 16$, converter o valor -73.538 do tipo `int` no tipo `unsigned short int`.

O valor original é reduzido módulo $2^{16} = 65.536$:

$$-73.538 + 65.536 + 65.536 = 57.534$$

O valor 57.354 do tipo `unsigned short int` é o resultado da redução.

Tipo inteiro em tipo inteiro

Exemplo

Considerando o tamanho do tipo `signed char` = 8, qual o resultado do programa ao lado?

```
#include <stdio.h>
int main(void) {
  int a = -896;
  signed char b;
  b = a;
  printf("%d -> %hhd\n", a, b);
  return 0;
}
```

Tipo inteiro em tipo inteiro

Exemplo

Considerando o tamanho do tipo `signed char` $= 8$, qual o resultado do programa ao lado?

```
#include <stdio.h>
int main(void) {
  int a = -896;
  signed char b;
  b = a;
  printf("%d -> %hhd\n", a, b);
  return 0;
}
```

Pelo padrão o resultado não é definido.
O compilador gcc adota a redução módulo 2^8, como uma extensão do padrão:

$$-896 + 256 + 256 + 256 = -128$$

Conversões expansivas

`signed char`	$\Rightarrow$	`short int`, `int`, `long int` ou `long long int`
`short int`	$\Rightarrow$	`int`, `long int` ou `long long int`
`int`	$\Rightarrow$	`long int` ou `long long int`
`long int`	$\Rightarrow$	`long long int`
`unsigned char`	$\Rightarrow$	`unsigned short int`, `unsigned int`, `unsigned long int` ou `unsigned long long int`
`unsigned short int`	$\Rightarrow$	`unsigned int`, `unsigned long int` ou `unsigned long long int`
`unsigned int`	$\Rightarrow$	`unsigned long int` ou `unsigned long long int`
`unsigned long int`	$\Rightarrow$	`unsigned long long int`

Tipo inteiro em tipo real de ponto flutuante

1. Se o valor puder ser representado exatamente no novo tipo, ele é adotado sem modificação.
2. Se o valor estiver na faixa de representação do novo tipo, mas não puder ser representado exatamente, o resultado é ou o valor representável imediatamente superior ou o imediatamente inferior (*dependente da implementação*).
3. Se o valor estiver fora da faixa de representação do novo tipo, o comportamento é indefinido.

Tipo inteiro em tipo real de ponto flutuante

Exemplo

Considerando que o tipo `float` possui expoente de 8 e mantissa de 23 dígitos, os três inteiros representáveis exatamente a partir de $2^{24} = 16.777.216$ são:

representação binária				
s	expoente	mantissa	valor decimal	
0	10010111	00...00	$= (1 + 0 \times 2^{-1} + \ldots + 0 \times 2^{-23}) \times 2^{24}$	$= 16.777.216$
0	10010111	00...01	$= (1 + 0 \times 2^{-1} + \ldots + 1 \times 2^{-23}) \times 2^{24}$	$= 16.777.218$
0	10010111	00...10	$= (1 + 0 \times 2^{-1} + \ldots + 1 \times 2^{-22}) \times 2^{24}$	$= 16.777.220$

Assim, a conversão do inteiro 16.777.217 em um valor do tipo `float` resultará no valor 16.777.216,0 ou no valor 16.777.218,0.

Tipo real de ponto flutuante em tipo inteiro

1. Se o valor da parte inteira pode ser representado no novo tipo, a parte fracionária é descartada e o valor do novo tipo é idêntico ao valor da parte inteira.
2. Se o valor da parte inteira não pode ser representado no novo tipo, o comportamento é indefinido.

Tipo real de ponto flutuante em tipo inteiro

Exemplo

Usando complemento-2 para representar os valores negativos e considerando o tamanho do tipo `short int` = 16 bits, Os valores desse tipo estão na faixa $[-32.768, 32.767]$.

Logo,

- O valor 32.600,234 do tipo `double` é convertido em 32.600 do tipo `short int`.
- A conversão do valor 40.000,00 do tipo `double` em um valor do tipo `short int` é indefinida.

Tipo real de ponto flutuante em tipo real de ponto flutuante

1. Se o valor puder ser representado exatamente no novo tipo, ele é adotado sem modificação.
2. Se o valor está na faixa de representação do novo tipo mas não pode ser representado exatamente, o resultado é ou o valor representável imediatamente superior ou o imediatamente inferior (*dependente da implementação*).
3. Se o valor está fora da faixa de representação do novo tipo, o resultado é indefinido.

Tipo real de ponto flutuante em tipo real de ponto flutuante

1. Se o valor puder ser representado exatamente no novo tipo, ele é adotado sem modificação.
2. Se o valor está na faixa de representação do novo tipo mas não pode ser representado exatamente, o resultado é ou o valor representável imediatamente superior ou o imediatamente inferior (*dependente da implementação*).
3. Se o valor está fora da faixa de representação do novo tipo, o resultado é indefinido.

As seguintes conversões são expansivas:

`float`	$\Rightarrow$	`double` ou `long double`
`double`	$\Rightarrow$	`long double`

Conversões envolvendo tipos complexos

Tipo real em tipo complexo

- O valor real é convertido na parte real do tipo complexo segundo as regras aplicáveis.
- A parte imaginária é zero (positivo ou não sinalizado).

Conversões envolvendo tipos complexos

Tipo real em tipo complexo

- O valor real é convertido na parte real do tipo complexo segundo as regras aplicáveis.
- A parte imaginária é zero (positivo ou não sinalizado).

Tipo complexo em tipo real

- A parte real do tipo complexo é convertida no valor real segundo as regras aplicáveis.
- A parte imaginária é descartada.

Conversões envolvendo tipos complexos

Tipo real em tipo complexo

- O valor real é convertido na parte real do tipo complexo segundo as regras aplicáveis.
- A parte imaginária é zero (positivo ou não sinalizado).

Tipo complexo em tipo real

- A parte real do tipo complexo é convertida no valor real segundo as regras aplicáveis.
- A parte imaginária é descartada.

Tipo complexo em tipo complexo

- As partes real e imaginária são convertidas segundo as regras aplicáveis aos tipos reais correspondentes.

Conversões envolvendo o tipo booliano

Tipo escalar em tipo `_Bool`

- Resulta no valor 0 se o valor original for igual a 0.
- Resulta no valor 1, em caso contrário.

Conversões envolvendo o tipo booliano

Tipo escalar em tipo `_Bool`

- Resulta no valor 0 se o valor original for igual a 0.
- Resulta no valor 1, em caso contrário.

Tipo `_Bool` em tipo escalar

- Resulta no valor original (0 ou 1), sem perda de informação.

Promoção inteira

Ocorre sempre que um valor de um tipo `int` ou `unsigned int` é esperado em alguma situação e o valor fornecido é

- de um tipo de menor ordem, ou
- um campo de bits do tipo `_Bool`, `int`, `signed int` ou `unsigned int`.

Regra

Se o valor puder ser representado como um valor do tipo `int`, ele é convertido em `int`. Caso contrário, ele é convertido em `unsigned int`.

Ordenamento de inteiros

O ordenamento dos tipos inteiros é utilizado na promoção inteira e nas conversões de valores.

- Os tipos inteiros sinalizados são ordenados pela precisão:
  `signed char` < `short int` < `int` < `long int` < `long long int`
- Os tipos inteiros não sinalizados possuem ordem igual à do tipo sinalizado correspondente:
  `unsigned char` < `unsigned short int` < `unsigned int` <
  `unsigned long int` < `unsigned long long int`
- O tipo `_Bool` possui ordem menor que a dos demais inteiros.
- A ordem do tipo `char` é igual à dos tipos `signed char` e `unsigned char`.
- A ordem dos tipos estendidos é dependente da implementação.

Compatibilidade de tipos

Os tipos em uma mesma linha são compatíveis — representam o mesmo tipo:

- `void`.
- `char`.
- `signed char`.
- `unsigned char`.
- `short`, `signed short`, `short int` e `signed short int`.
- `unsigned short` e `unsigned short int`.
- `int`, `signed` e `signed int`.
- `unsigned` e `unsigned int`.
- `long`, `signed long`, `long int` e `signed long int`.
- `unsigned long` e `unsigned long int`.
- `long long`, `signed long long`, `long long int` e `signed long long int`.
- `unsigned long long` e `unsigned long long int`.
- `float`.
- `double`.
- `long double`.
- `_Bool`.
- `float _Complex`.
- `double _Complex`.
- `long double _Complex`.

Tipos predefinidos

Tipo	Finalidade	Declaração
bool	Tipo booliano idêntico a _Bool.	stdbool.h
size_t	Tipo inteiro não sinalizado, representa tamanhos.	stddef.h stdlib.h wchar.h
ptrdiff_t	Tipo inteiro sinalizado, representa diferença entre ponteiros.	stddef.h
intmax_t	Tipo do maior inteiro sinalizado.	stdint.h
uintmax_t	Tipo do maior inteiro não sinalizado.	stdint.h
wchar_t	Tipo inteiro usado para representar caracteres estendidos.	stddef.h stdlib.h wchar.h

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Literais e constantes

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Literais inteiros

Os literais inteiros exprimem números inteiros como 1, 234 e 666.

Sem prefixo $\Rightarrow$ valor decimal.

Prefixo `0` $\Rightarrow$ valor octal.

Prefixo `0x` ou `0X` $\Rightarrow$ valor hexadecimal.

Literais inteiros

Os literais inteiros exprimem números inteiros como 1, 234 e 666.

Sem prefixo $\Rightarrow$ valor decimal.

Prefixo `0` $\Rightarrow$ valor octal.

Prefixo `0x` ou `0X` $\Rightarrow$ valor hexadecimal.

Exemplo

Literal inteiro	Valor decimal		
`-12`	-12		
`354`	354		
`-012`	-10	$=$	$-(1 \times 8^1 + 2 \times 8^0)$
`0354`	236	$=$	$3 \times 8^2 + 5 \times 8^1 + 4 \times 8^0$
`-0x12`	-18	$=$	$-(1 \times 16^1 + 2 \times 16^0)$

Tipo dos literais inteiros

Sufixos

- `u` ou `U` ⇒ Literal do tipo `unsigned int`.
- `l` ou `L` ⇒ Literal do tipo `long int`.
- `ll` ou `LL` ⇒ Literal do tipo `long long int`.

Tipo dos literais inteiros

Sufixos

- `u` ou `U` ⇒ Literal do tipo `unsigned int`.
- `l` ou `L` ⇒ Literal do tipo `long int`.
- `ll` ou `LL` ⇒ Literal do tipo `long long int`.

Exemplo

`12u`	Tipo `unsigned int`; representa o valor 12.
`3Ul`	Tipo `unsigned long int`; representa o valor 3.
`0x45LL`	Tipo `long long int`; representa o valor 69.
`071ull`	Tipo `unsigned long long int`; representa o valor 57.

Tipo dos literais inteiros

O uso do sufixo apenas orienta a determinação do tipo

Sufixo	Valor decimal	Valor octal ou hexadecimal

	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`l` ou `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`ll` ou `LL`	`long long int`	`long long int` `unsigned long long int`

Tipo dos literais inteiros

Exemplo

Considerando a representação de negativos em complemento-2, os tamanhos de 32 bits para os tipos `int` e `long int` e de 64 bits para o tipo `long long int`, determine o tipo dos seguintes literais:

Literal inteiro	Tipo
67	
2147483648	
2147483648uL	
040000000000	
0x8000000000000000	

Tipo dos literais inteiros

Exemplo

Considerando a representação de negativos em complemento-2, os tamanhos de 32 bits para os tipos `int` e `long int` e de 64 bits para o tipo `long long int`, determine o tipo dos seguintes literais:

Literal inteiro	Tipo
67	`int`
2147483648	
2147483648uL	
040000000000	
0x8000000000000000	

Tipo dos literais inteiros

Exemplo

Considerando a representação de negativos em complemento-2, os tamanhos de 32 bits para os tipos `int` e `long int` e de 64 bits para o tipo `long long int`, determine o tipo dos seguintes literais:

Literal inteiro	Tipo
67	`int`
2147483648	`long long int`
2147483648uL	
040000000000	
0x8000000000000000	

Tipo dos literais inteiros

Exemplo

Considerando a representação de negativos em complemento-2, os tamanhos de 32 bits para os tipos `int` e `long int` e de 64 bits para o tipo `long long int`, determine o tipo dos seguintes literais:

Literal inteiro	Tipo
67	`int`
2147483648	`long long int`
2147483648uL	`unsigned long int`
040000000000	
0x8000000000000000	

Tipo dos literais inteiros

Exemplo

Considerando a representação de negativos em complemento-2, os tamanhos de 32 bits para os tipos `int` e `long int` e de 64 bits para o tipo `long long int`, determine o tipo dos seguintes literais:

Literal inteiro	Tipo
67	`int`
2147483648	`long long int`
2147483648uL	`unsigned long int`
040000000000	`long long int`
0x8000000000000000	

Tipo dos literais inteiros

Exemplo

Considerando a representação de negativos em complemento-2, os tamanhos de 32 bits para os tipos `int` e `long int` e de 64 bits para o tipo `long long int`, determine o tipo dos seguintes literais:

Literal inteiro	Tipo
67	`int`
2147483648	`long long int`
2147483648uL	`unsigned long int`
040000000000	`long long int`
0x8000000000000000	`unsigned long long int`

Literais reais

Os literais reais são identificados

- Pelo uso do ponto decimal: `23.`, `0.34`, `12.6`
- Pela notação científica: `2.3E1`, `34e-2`, `.126E2`

Notação científica decimal

Exprime o número como uma potência de 10, com o termo `E` ou `e` introduzindo o expoente:

$$\langle \mathit{Significando} \rangle \langle E \rangle \langle \mathit{Expoente} \rangle$$

O valor decimal é dado por

$$\langle \mathit{Significando} \rangle \times 10^{\langle \mathit{Expoente} \rangle}$$

Notação científica decimal

Exprime o número como uma potência de 10, com o termo `E` ou `e` introduzindo o expoente:

$$\langle \mathit{Significando} \rangle \langle E \rangle \langle \mathit{Expoente} \rangle$$

O valor decimal é dado por

$$\langle \mathit{Significando} \rangle \times 10^{\langle \mathit{Expoente} \rangle}$$

Exemplo

$$\begin{aligned} \texttt{2.3E1} &= 2,3 \times 10^1 = 23,0 \\ \texttt{34e-2} &= 34 \times 10^{-2} = 0,34 \\ \texttt{.126E2} &= 0,126 \times 10^2 = 12,6 \end{aligned}$$

Notação científica hexadecimal

Exprime o número como uma potência de 2, na base hexadecimal, com o termo `P` ou `p` introduzindo o expoente:

$$\langle \textit{Prefixo-hex} \rangle \langle \textit{Significando-hex} \rangle \langle \textit{P} \rangle \langle \textit{Expoente} \rangle$$

O valor decimal é dado por

$$(\text{Valor decimal do significando}) \times 2^{\langle \textit{Expoente} \rangle}$$

Notação científica hexadecimal

Exemplo

Literais reais hexadecimais	Valor representado
`0x1Ap2`	$104,0 = (1 \times 16^1 + 10 \times 16^0) \times 2^2$
`0Xd0P-1`	$104,0 = (13 \times 16^1 + 0 \times 16^0) \times 2^{-1}$
`0x5.2p-2`	$1,28125 = (5 \times 16^1 + 2 \times 16^{-1}) \times 2^{-2}$
`0X0.29P3`	$1,28125 = (0 \times 16^0 + 2 \times 16^{-1} + 9 \times 16^{-2}) \times 2^3$

Tipo dos literais reais

Todo literal real é do tipo `double`, exceto se possuir o sufixo

- `F` ou `f`, indicando o tipo `float`
- `L` ou `l`, indicando o tipo `long double`.

Exemplo

Literal	Tipo
98E-1	double
98E-1F	float
0.2L	long double

Literais caracteres

Escritos entre aspas simples, representam caracteres.

Exemplo

'a'	letra a
' '	espaço
'2'	dígito dois

Literais caracteres

Quando o literal é interpretado como um inteiro, obtém-se o código numérico do caractere como um valor do tipo `int`.

Exemplo

O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  char c = 'a';
  printf("%c %hhd\n", c, c);
  return 0;
}
```

Literais caracteres

Quando o literal é interpretado como um inteiro, obtém-se o código numérico do caractere como um valor do tipo `int`.

Exemplo

O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  char c = 'a';
  printf("%c %hhd\n", c, c);
  return 0;
}
```

Resposta: a 97

Caracteres especiais

`\b`	recuo de posição
`\t`	avanço tabulação horizontal
`\v`	avanço tabulação vertical
`\n`	nova linha
`\r`	retorno de carro
`\f`	avanço de formulário
`\a`	alarme
`\"`	aspa dupla
`\'`	aspa simples
`\\`	barra invertida
`\?`	interrogação

Literais caracteres - representação octal

Os caracteres podem ser expressos como `\ddd`, onde `ddd` são dígitos octais representando um valor do tipo `unsigned char`.

Exemplo

`'\153'`	corresponde ao caractere k
`'\046'`	corresponde ao caractere &
`'\12'`	corresponde ao caractere LF (nova linha)

Literais caracteres - representação hexadecimal

Os caracteres podem ser expressos como `\xdd`, onde `dd` são dígitos hexadecimais representando um valor do tipo `unsigned char`.

Exemplo

`'\x6B'`	corresponde ao caractere k
`'\x26'`	corresponde ao caractere &
`'\xa'`	corresponde ao caractere LF (nova linha)

Literais caracteres - representação Unicode

Os caracteres podem ser expressos como `\udddd`, onde `dddd` são dígitos hexadecimais representando o código Unicode do caractere.

- Os literais Unicode não podem conter código na faixa de `\uD800` a `\uDFFF`.
- Os literais Unicode não podem conter código menores que `\u00A0`, exceto os códigos `\u0024`, `\u0040` e `\u0060`.

Exemplo

`'\u0024'`	corresponde ao caractere $
`'\u0040'`	corresponde ao caractere @
`'\u0060'`	corresponde ao caractere '

Literais caracteres multibytes

Se um literal caractere contém um caractere que não faz parte do conjunto básico dos caracteres de execução ou quando contém dois ou mais caracteres, ele é interpretado como um *caractere multibyte*.

Exemplo

`'olas'` é interpretado como um caractere multibyte.

Os caracteres multibytes não devem ser usados como se fossem caracteres básicos.

Literais caracteres estendidos

Os literais que representam caracteres estendidos devem ter prefixo

- `L`,
- `u`, quando é adotada a codificação Unicode `UTF-16`, ou
- `U`, quando é adotada a codificação Unicode `UTF-32`.

Exemplo

O literal `L'\x456'` representa corretamente o caractere estendido cujo código decimal é 1.110.

A representação gráfica desse caractere depende da localização em vigor no ambiente de execução.

Tipo dos literais caracteres

- O tipo de um literal caractere é `int`
- O tipo de um literal caractere estendido é
  - `wchar_t`
  - `char16_t`, para codificação Unicode UTF-16
  - `char32_t`, para codificação Unicode UTF-32

O tipo `wchar_t` é declarado no cabeçalho `stddef.h`.
Os tipos `char16_t` e `char32_t`, declarados no cabeçalho `uchar.h`, foram incluídos na versão 2011 do padrão da linguagem.

Tipo dos literais caracteres

- O tipo de um literal caractere é `int`
- O tipo de um literal caractere estendido é
  - `wchar_t`
  - `char16_t`, para codificação Unicode `UTF-16`
  - `char32_t`, para codificação Unicode `UTF-32`

O tipo `wchar_t` é declarado no cabeçalho `stddef.h`.
Os tipos `char16_t` e `char32_t`, declarados no cabeçalho `uchar.h`, foram incluídos na versão 2011 do padrão da linguagem.

Os literais que representam caracteres estendidos devem ser armazenados em variáveis do tipo apropriado:

- `wchar_t` para literais com prefixo `L`
- `char16_t` para literais com prefixo `u`
- `char32_t` para literais com prefixo `U`

Literais cadeia de caracteres

Os literais cadeia de caracteres são escritos entre aspas duplas.

Exemplo

``456.3''	Cadeia com 5 caracteres
``53''	Cadeia com 2 caracteres
``x''	Cadeia com 1 caractere

Literais cadeia de caracteres

Os caracteres em uma cadeia de caracteres podem ser expressos das formas já vistas.

Exemplo

O que é impresso pelo programa abaixo?

```
#include<stdio.h>
int main (void){
printf("V\x61lor\x20\075 R\u0024 23.7\b\b,88\b0.");
return 0;
}
```

Literais cadeia de caracteres

Os caracteres em uma cadeia de caracteres podem ser expressos das formas já vistas.

Exemplo

O que é impresso pelo programa abaixo?

```
#include<stdio.h>
int main (void){
printf("V\x61lor\x20\075 R\u0024 23.7\b\b,88\b0.");
return 0;
}
```

Resposta: `Valor = R$ 23,80.`

Cadeias estendidas de caracteres

- Quando um caractere da cadeia não faz parte do conjunto básico dos caracteres de execução a cadeia é considerada estendida.
- Os literais cadeias estendidas de caracteres devem ser expressos com os prefixos
  - `L`, se a cadeia possui caracteres estendidos,
  - `u8`, se os caracteres da cadeia são `UTF-8`,
  - `u`, se os caracteres da cadeia são `UTF-16`, ou
  - `U`, se os caracteres da cadeia são `UTF-32`.

  Exemplos: `L‘‘ação’’` e `L‘‘ósculo’’`.
- Existem funções próprias para lidar com cadeias de caracteres estendidas.

Os prefixos `u8`, `u` e `U` foram incluídos na versão 2011 do padrão da linguagem.

Tipo dos literais cadeias de caracteres

- O tipo de um literal cadeia de caracteres é
  - `char *`, se a cadeia não tem prefixo ou tem prefixo `u8`.
- O tipo de um literal cadeia estendida de caracteres é
  - `wchar_t *`, se a cadeia tem prefixo `L`,
  - `char16_t *`, se a cadeia tem prefixo `u`, ou
  - `char32_t *`, se a cadeia tem prefixo `U`.

Literais compostos

Os literais compostos são expressões usadas para atribuir valor a variáveis do tipo por eles especificado.

⟨*LiteralComposto*⟩ ::= **(** ⟨*NomeTipo*⟩ **)** **{** ⟨*ListaIniciação*⟩ **}**

- Um literal composto cria em memória um objeto não nomeado do tipo especificado, iniciando-o com os valores da lista de iniciação.
- O tipo de um literal composto é o especificado em sua expressão ou o derivado da sua lista de iniciação, se ele for especificado como um vetor de tamanho indefinido.

Literais compostos

Exemplo

`(struct r_aluno) {"Josefa, linda e bela", 'f', 453}`
: Cria uma estrutura do tipo `struct r_aluno` atribuindo ao seu primeiro componente a cadeia "Josefa, linda e bela"; ao segundo, o caractere 'f'; e ao terceiro, o inteiro 453.

`(int){5712}`
: Cria um objeto do tipo `int` e o inicia com o valor 5.712.

`(char []) {"E o anjo torna:  - A Morte sou!"}`
: Cria um vetor de caracteres iniciando-o com os caracteres da cadeia "E o anjo torna: — A Morte sou!".

Enumerações

As enumerações são listas de constantes declaradas por meio da palavra-chave `enum`.

$\langle \textit{Enumeração} \rangle$ ::= **enum** [$\langle \textit{Etiqueta} \rangle$] **{** $\langle \textit{ListaEnum} \rangle$ [,] **}**

$\langle \textit{ListaEnum} \rangle$ ::= $\langle \textit{CteEnumerada} \rangle$ | $\langle \textit{ListaEnum} \rangle$ **,** $\langle \textit{CteEnumerada} \rangle$

$\langle \textit{CteEnumerada} \rangle$ ::= $\langle \textit{Identificador} \rangle$ | $\langle \textit{Identificador} \rangle$ = $\langle \textit{ExprCte} \rangle$

Enumerações

Cada constante enumerada é um identificador associado a um valor fixo:

- O primeiro assume o valor de sua expressão de atribuição ou o valor zero, se ela não existir.
- Os demais assumem o valor de sua expressão de atribuição ou o valor do identificador anterior incrementado de 1, se ela não existir.

Enumerações

Cada constante enumerada é um identificador associado a um valor fixo:

- O primeiro assume o valor de sua expressão de atribuição ou o valor zero, se ela não existir.
- Os demais assumem o valor de sua expressão de atribuição ou o valor do identificador anterior incrementado de 1, se ela não existir.

Exemplo

As seguintes enumerações são válidas:

```
enum {zero, um, dois, tres, quatro, cinco}
enum {pre, nor = 4, reg = 4, sup, exc}
enum naipe {ouros = 1, copas, paus, espadas}
```

Tipo das enumerações

1. O tipo de cada constante enumerada é `int`.
2. Cada enumeração define um tipo próprio diferente dos demais.
3. O tipo de uma enumeração pode ser referido por meio da sua etiqueta.

Tipo das enumerações

1. O tipo de cada constante enumerada é `int`.
2. Cada enumeração define um tipo próprio diferente dos demais.
3. O tipo de uma enumeração pode ser referido por meio da sua etiqueta.

- O uso de etiqueta permite declarar variáveis do tipo enumerado especificado pela etiqueta.
- Em princípio, as variáveis declaradas dessa forma deveriam assumir apenas os valores da enumeração.

Exemplo

A declaração `enum naipe carta;` declara a variável `carta` como do tipo enumerado `enum naipe`.

Variáveis constantes

O uso do qualificador `const` faz com que o valor da variável não possa ser modificado.

```
const int per;
float const sal, taxa = 3.2F;
```

Na ilustração acima, as constantes `per` e `sal` não possuem valor de iniciação. Assumem um valor indefinido ou o zero dependendo do contexto em que são declaradas.

Variáveis constantes

O uso do qualificador `const` faz com que o valor da variável não possa ser modificado.

```
const int per;
float const sal, taxa = 3.2F;
```

Na ilustração acima, as constantes `per` e `sal` não possuem valor de iniciação. Assumem um valor indefinido ou o zero dependendo do contexto em que são declaradas.

Observação!

As variáveis qualificadas como `const` não são expressões constantes.

Macros

As macros são nomes associados a uma expressão por meio da construção #define.

```
#define g 9.8
#define taxa 2.3
#define pi (3.1415)
```

Macros

As macros são substituídas por suas expressões durante o pré-processamento.

Macros

As macros são substituídas por suas expressões durante o pré-processamento.

Programa original

```
#include <stdio.h>
#define PI (3.1415)
#define Sigla "br"
int main(void) {
  #define G (9.89)
  printf("pi = %f ", 2 * PI);
  printf("g = %f\n", G);
  printf("%s\n", Sigla);
  return 0;
}
```

Macros

As macros são substituídas por suas expressões durante o pré-processamento.

Programa após o pré-processamento

```
#include <stdio.h>
int main(void) {
  printf("pi = %f ", 2 * (3.1415));
  printf("g = %f\n", (9.89));
  printf("%s\n", "br");
  return 0;
}
```

Literais boolianos

Os nomes `true` e `false` são definidos como macros no arquivo-cabeçalho `stdbool.h`:

```
#define true 1
#define false 0
```

Rótulos

- Os rótulos são identificadores seguidos imediatamente de dois pontos.
- São colocados antes de um comando, e
- rotulam o comando que os segue.

Exemplo

```
#include <stdio.h>
int main(void) {
  int a, b = 2;
rot1: printf("Digite um inteiro: ");
  scanf("%d", &a);
rot2:
rot3:
  if (a > b) {
     printf("%d > ", a); rot4: printf("%d\n", b);
  }
  return 0;
}
```

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Identificadores e variáveis

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Palavras-chaves

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	inline	int
long	register	restrict	return	short	signed
sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while		

_Alignas _Alignof _Atomic _Bool
_Complex _Generic _Imaginary _Noreturn
_Static_assert _Thread_local

Identificadores

- Sequências não nulas de caracteres, iniciadas com um caractere diferente de dígito.
- Caracteres válidos:
  - dígitos,
  - letras latinas maiúsculas e minúsculas e
  - sublinhado (_).
- Sensíveis à grafia.
- Não podem ser palavras-chave.

Classificação dos identificadores

Cada identificador pertence a uma classe que determina a natureza dos objetos por ele designados.

Rótulos. Identificadores usados como rótulos para nomear comandos.

Etiquetas. Identificadores usados para nomear as etiquetas de estruturas, uniões e enumerações.

Componentes. Identificadores usados para nomear os componentes de estruturas e uniões.

Ordinários. Todos os demais identificadores, incluindo aqueles usados para nomear variáveis que não são componentes de estruturas ou uniões.

Declarando variáveis

- As variáveis são identificadores usados para designar uma localização específica da memória.
- Toda variável deve ser
  - Declarada antes de ser referenciada.
  - Definida (alocada) antes de ser usada.

⟨*DeclaraçãoVar*⟩	::=	⟨*DeclTipo*⟩ [⟨*ListaDeclVar*⟩] **;**
⟨*DeclTipo*⟩	::=	⟨*EspecTipo*⟩ [⟨*DeclTipo*⟩]
	|	⟨*ClasseArmz*⟩ [⟨*DeclTipo*⟩]
	|	⟨*QualifTipo*⟩ [⟨*DeclTipo*⟩]
	|	⟨*EspecAlin*⟩ [⟨*DeclTipo*⟩]
⟨*EspecTipo*⟩	::=	**void** | **char** | **short** | **int** | **long** | **float** | **double** | **signed** | **unsigned** | **_Bool** | **_Complex** | **_Atomic (**⟨*NomeTipo*⟩**)**
⟨*ClasseArmz*⟩	::=	**extern** | **static** | **auto** | **register** | **_Thread_local**
⟨*QualifTipo*⟩	::=	**const** | **restrict** | **volatile** | **_Atomic**
⟨*EspecAlin*⟩	::=	**_Alignas (**⟨*BaseAlin*⟩**)**
⟨*ListaDeclVar*⟩	::=	⟨*DeclVar*⟩ | ⟨*ListaDeclVar*⟩ **,** ⟨*DeclVar*⟩
⟨*DeclVar*⟩	::=	[⟨*DeclPonteiro*⟩] ⟨*Declarador*⟩
	|	[⟨*DeclPonteiro*⟩] ⟨*Declarador*⟩ **=** ⟨*ExprInic*⟩
⟨*Declarador*⟩	::=	⟨*Identificador*⟩ | ⟨*DeclaradorVetor*⟩ | ⟨*DeclaradorFunção*⟩
	|	**(** [⟨*DeclPonteiro*⟩] ⟨*Declarador*⟩ **)**
⟨*Identificador*⟩	::=	Identificador válido

Declarando variáveis

Exemplo

```
int val;
int valor, taxa = 23;
extern char sexo;
const long double G = 9.8, veloc;
register const volatile unsigned int media;
```

Escopo dos identificadores

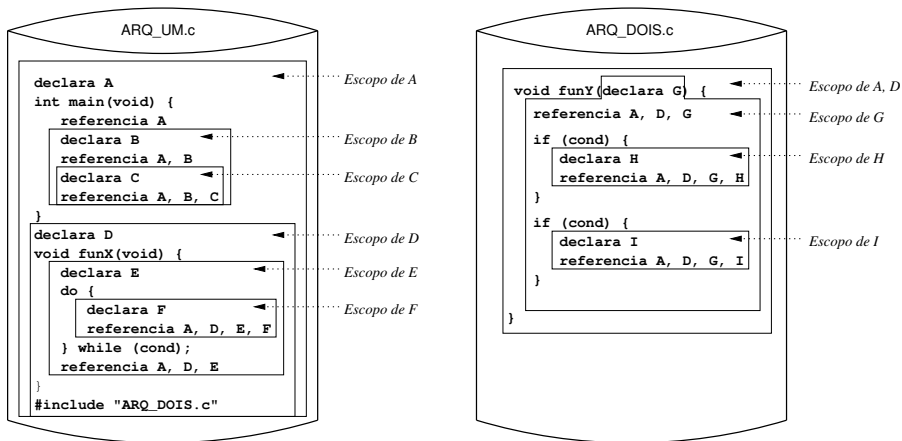
Trecho do programa no qual o identificador pode ser diretamente referido.

Função. Podem ser referidos em qualquer ponto da função onde ocorrem.

Protótipo de função. Podem ser referidos apenas na declaração da função.

Bloco. Podem ser referidos do ponto da declaração até o fim do bloco no qual são declarados. Se a declaração ocorrer na lista de parâmetros de uma função, o escopo vai até o fim do bloco que delimita a função.

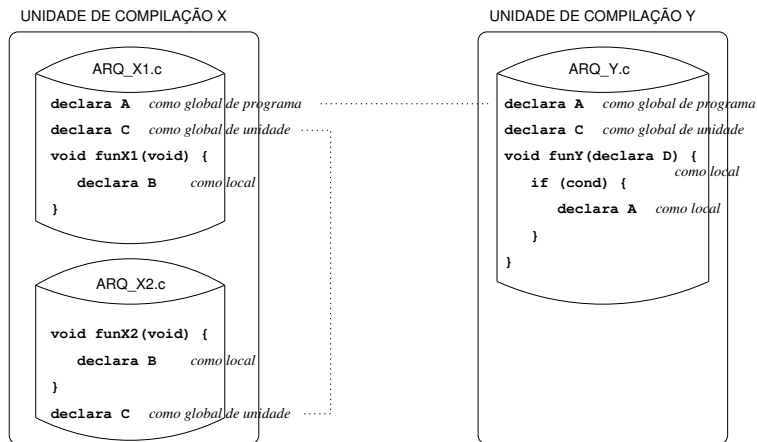
Arquivo. Podem ser referidos do ponto da declaração até o fim da unidade de compilação onde são declarados.

Escopo dos identificadores

Variáveis globais e locais

Locais. As variáveis com escopo de bloco são locais (ao bloco em que são declaradas).

Globais. As variáveis com escopo de arquivo são globais.

- **Globais de programa.** Quando a variável pode ser referida em várias unidades de compilação.
- **Globais de unidade.** Quando a variável pode ser referida apenas na unidade de compilação onde foi declarada.

Variáveis globais e locais

Ligação dos identificadores

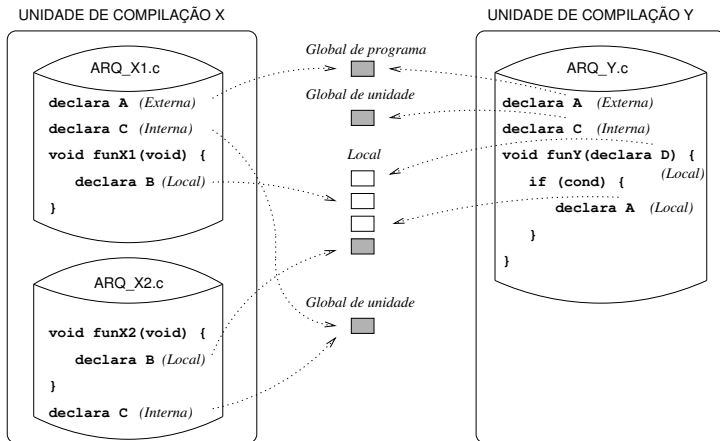
- Variáveis de nomes distintos designam um espaço de memória diferente dos demais.
- Variáveis de mesmo nome podem designar o mesmo espaço de memória ou não, dependendo do escopo e do modo de ligação estabelecido entre os seus identificadores.

Ligação dos identificadores

Externa. Denotam o mesmo objeto em todas as unidades de compilação que compõem o programa.

Interna. Denotam o mesmo objeto, dentro de uma mesma unidade de compilação.

Local. Denotam um espaço de memória próprio. (*As variáveis com ligação local também são chamadas de variáveis sem ligação*).

Ligação dos identificadores

Alocação de memória

O modo de alocação de uma variável determina o momento em que a alocação ocorre, sua duração e a atribuição do valor inicial ao espaço alocado.

Estático. A variável é alocada uma única vez, antes do início da execução do programa, e permanece alocada durante toda a execução.
Valor inicial: valor padrão ou o valor da expressão de iniciação.

Automático. A variável é alocada sempre que a execução do programa inicia o bloco no qual ela é declarada, permanecendo alocada até que o bloco seja finalizado.
Valor inicial: indeterminado ou o valor da expressão de iniciação.

Por comando. A alocação ocorre em decorrência da execução de comandos próprios de alocação de memória.

Ciclo de vida de uma variável

Parcela do tempo de execução do programa que vai da alocação da variável até o momento em que é desalocada.

Variáveis estáticas. Compreende toda a execução do programa.

Variáveis automáticas.

Se o tipo não é um vetor variável.

- São alocadas no início do bloco.
- O ciclo de vida compreende a execução do bloco em que são declaradas.

Se o tipo é um vetor variável.

- São alocadas na declaração.
- O ciclo de vida compreende a execução do trecho de programa que corresponde ao seu escopo.

Declaração e definição

- Uma mesma variável pode ser declarada várias vezes no texto de um programa.
    - Isso permite a referência à variável em cada unidade de compilação onde é declarada.
- As variáveis devem ser definidas apenas uma vez.

Declaração e definição

- Uma mesma variável pode ser declarada várias vezes no texto de um programa.
  - Isso permite a referência à variável em cada unidade de compilação onde é declarada.
- As variáveis devem ser definidas apenas uma vez.

Declaração. Introduz o nome da variável, com seu tipo, qualificações e classe de armazenamento.

Definição. É a declaração que (quando executada) causa alocação de memória à variável.

Declaração e definição

São definições:

- Declaração de variáveis com escopo de bloco.
- Declaração de variáveis com escopo de arquivo, com expressão de iniciação.

São definições provisórias:

- Declarações de variáveis com escopo de arquivo, sem expressão de iniciação,
  - que não especificam a classe de armazenamento ou
  - que são explicitamente declaradas como `static`.

Declaração e definição

- As variáveis com ligação externa devem ter no máximo uma definição em todo o programa.
- As variáveis com ligação interna devem ter no máximo uma definição na unidade de compilação na qual são declaradas.

Classe de armazenamento

A classe de armazenamento e o escopo determinam a ligação e o modo de alocação das variáveis.

`extern` Modo de alocação estático.

- Ligação interna, se houver no mesmo escopo uma variável (visível) de mesmo nome declarada com ligação interna.
- Ligação externa, em caso contrário.

`static` Modo de alocação estático.

- Ligação interna, se tem escopo de arquivo.
- Ligação local, se tem escopo de bloco.

`auto` Modo de alocação automático. Ligação local. O escopo desse tipo de variável deve ser de bloco.

`register` Modo de alocação automático. Ligação local. O escopo desse tipo de variável deve ser de bloco.

Classe de armazenamento

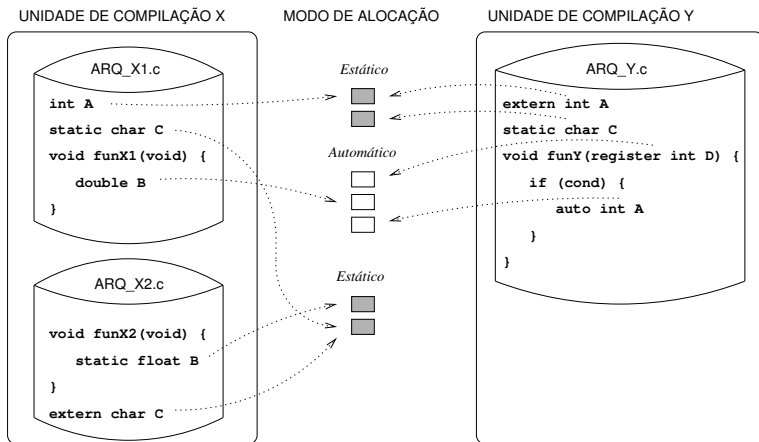
Sem qualificador. Uma variável declarada sem classe de armazenamento, possui

- Modo de alocação estático e ligação externa, se tem escopo de arquivo.
- Modo de alocação automático e ligação local, se tem escopo de bloco.

Classe de armazenamento

Armazenamento	Escopo	Ligação	Alocação
`extern`	arquivo	externa	estático
	ou bloco	interna*	
`static`	arquivo	interna	estático
	bloco	local	
`auto`	bloco	local	automático
`register`	bloco	local	automático
Sem qualificador	arquivo	externa	estático
	bloco	local	automático

*Apenas se existir variável visível com ligação interna.

Classe de armazenamento

Qualificadores de tipo

Determinam o modo como as variáveis são modificadas.

`const` Indica que a variável não pode ter seu conteúdo modificado.

`restrict` Indica que o espaço de memória designado por um ponteiro só pode ser acessado através desse ponteiro, ou de endereços baseados nesse ponteiro.

`volatile` Indica que o conteúdo da memória designada pela variável pode ser modificado por ações externas ao programa como, por exemplo, por outros programas ou por mecanismos de hardware.

Valores iniciais

Variáveis estáticas com expressão de iniciação

- O valor inicial é o valor da expressão.
- A expressão deve ser constante.

Variáveis estáticas sem expressão de iniciação

- Ponteiro nulo, se a variável é do tipo ponteiro.
- Zero, se a variável é de um tipo aritmético.
- Os componentes de tipos agregados (vetores e estruturas) são recursivamente iniciados segundo estas regras.
- O primeiro componente de uma união é recursivamente iniciado segundo estas regras.

Variáveis automáticas são iniciadas com o valor da sua expressão de iniciação, se houver, ou com um valor indeterminado, em caso contrário.

Valores iniciais

Variáveis com escopo de bloco e declaradas com o especificador `extern` não podem ser iniciadas.

Expressões constantes

- As expressões constantes são avaliadas em tempo de compilação.
- Não podem conter operador de atribuição, incremento, decremento, vírgula, nem chamadas a funções.

Expressões constantes

As expressões constantes podem ser:

Inteiras. Expressões contendo apenas operandos que são constantes inteiras, enumeradas ou caracteres, constantes reais convertidas em um tipo inteiro, ou expressões com o operador sizeof que resultem em um tipo inteiro.

Aritméticas. Expressões contendo apenas operandos que são constantes reais, inteiras, enumeradas ou caracteres, ou expressões com o operador sizeof.

de endereço. Pode ser o ponteiro nulo, ponteiros para um objeto que tenha alocação estática ou ponteiros para função.

Valores não identificados

Existem valores criados em memória sem identificação:

- Literais do tipo cadeia de caracteres
- Literais compostos

Ocultação de variáveis

Duas variáveis com o mesmo nome não podem coexistir se o escopo de uma termina exatamente no mesmo ponto que o escopo da outra.

Exemplo

O programa ao lado está errado. As variáveis `val` não podem coexistir.

```
#include <stdio.h>
int main(void) {
  int aux = 25;
  int val = 23;
  printf("%d\n", val);
  float val = 40.0;
  printf("%f\n",
              val+aux);
  return 0;
}
```

Ocultação de variáveis

Exemplo

Os dois programas abaixo estão corretos. As variáveis val coexistem, com uma ocultando a outra.

```
#include <stdio.h>
int main(void) {
  int aux = 25;
  if (aux > 10) {
    int val = 23;
    printf("%d\n", val);
  }
  if (aux > 5) {
    float val = 40.0;
    printf("%f\n",
                val+aux);
  }
  return 0;
}
```

```
#include <stdio.h>
int main(void) {
  int aux = 25;
  int val = 23;
  printf("%d\n", val);
  if (val > 10) {
    float val = 40.0;
    printf("%f\n",
                val+aux);
  }
  return 0;
}
```

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Operadores e expressões

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Classificação

Quantidade de operandos

- Unário
- Binário
- Ternário

Classificação

Quantidade de operandos

- Unário
- Binário
- Ternário

Notação

- Prefixada
- Pós-fixada
- Infixada

Classificação

Quantidade de operandos

- Unário: `-a`
- Binário: `a * b`
- Ternário: `a ? b : c`

Notação

- Prefixada: `&a`
- Pós-fixada: `a++`
- Infixada: `a + b`

Operadores aritméticos

Operação	Operador
Mais e menos unário	+ e -
Multiplicação	*
Divisão	/
Resto (Mod)	%
Adição e subtração	+ e -
Incremento e decremento	++ e --

Operadores aritméticos

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  int a, b, c, d;
  a = 7; b = -6;
  a = -a;
  b = +b;
  c = 2 - -2;
  d = 1 - -b;
  printf("%d %d %d %d\n", a, b, c, d);
  return 0;
}
```

Operadores aritméticos

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  int a, b, c, d;
  a = 7; b = -6;
  a = -a;
  b = +b;
  c = 2 - -2;
  d = 1 - -b;
  printf("%d %d %d %d\n", a, b, c, d);
  return 0;
}
```

Resposta: -7 -6 4 -5

Operadores aritméticos

Operações reais e inteiras

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  int a, b; double c, d; float e, f;
  a = 3 * 2;
  b = 9 / a;
  c = 3 * 2.0;
  d = 9.0 / a;
  e = 2 + 3 * (a / 2.0F);
  f = 2 * e;
  printf("%d %d %f\n", a, b, c);
  printf("%f %f %f\n", d, e, f);
  return 0;
}
```

Operadores aritméticos

Operações reais e inteiras

Exemplo

O que é impresso pelo seguinte programa?

Resposta:
```
6 1 6.000000
1.500000 11.000000 22.000000
```

Operadores aritméticos

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  int i = 200, j = 100, k = 2;
  printf("%d\n", ++i);
  printf("%d\n", i);
  printf("%d\n", j++);
  printf("%d\n", j);
  k = k + ++j + i++;
  printf("%d %d %d\n", i, j, k);
  i++; ++j;
  printf("%d %d %d\n", i, j, k);
  return 0;
}
```

Operadores aritméticos

Exemplo

O que é impresso pelo seguinte programa?

Resposta:

```
201
201
100
101
202  102  305
203  103  305
```

Valores especiais

Operação		Valor resultante
$\pm\texttt{inf} \times \pm\texttt{inf}$	$=$	$\pm\texttt{inf}$
$\pm\langle \mathit{Número} \rangle \div 0$	$=$	$\pm\texttt{inf}$
$\pm 0 \div \pm 0$	$=$	NAN
$\pm\texttt{inf} \div \pm\texttt{inf}$	$=$	NAN
$\pm\texttt{inf} \times 0$	$=$	NAN
$\texttt{inf} - \texttt{inf}$	$=$	NAN
$\langle \mathit{Número} \rangle \div \pm\texttt{inf}$	$=$	± 0

Valores especiais

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
#include <float.h>
int main(void) {
  double a, b, c, d, e, f, g, h, i;
  a = DBL_MIN / DBL_MAX;
  b = 4 / 0.0;
  c = -DBL_MIN / 0.0;
  d = b * c;
  e = b / c;
  f = DBL_MAX / b;
  g = DBL_MAX * b;
  h = b + c;
  i = 0.0 / 0.0;
  printf("%f %f %f\n", a, b, c);
  printf("%f %f %f\n", d, e, f);
  printf("%f %f %f\n", g, h, i);
  return 0;
}
```

Valores especiais

Exemplo

O que é impresso pelo seguinte programa?

Resposta:

```
0.000000 inf -inf
-inf nan 0.000000
inf nan nan
```

Operações de tipo complexo

Para tipos números complexos $x = a + bi$ e $y = c + di$:

$$
\begin{aligned}
x + y &= (a + c) + (b + d)i \\
x - y &= (a - c) + (b - d)i \\
x * y &= (ac - bd) + (ad + bc)i \\
x/y &= k, \text{ tal que } x = k * y, \; y \neq 0
\end{aligned}
$$

Operações de tipo complexo

Exemplo. O que é impresso pelo seguinte programa?

```
#include <stdio.h>
#include <complex.h>
int main(void) {
  double _Complex x = 2.0 + 3.0i;
  double _Complex y = 5.0 + 4.0i;
  double _Complex a = x + y;
  double _Complex b = x - y;
  double _Complex c = x * y;
  double _Complex d = x / y;
  printf("(%f + %fi)\n",
                   creal(a), cimag(a));
  printf("(%f + %fi)\n",
                   creal(b), cimag(b));
  printf("(%f + %fi)\n",
                   creal(c), cimag(c));
  printf("(%f + %fi)\n",
                   creal(d), cimag(d));
  return 0;
}
```

Operações de tipo complexo

Exemplo. O que é impresso pelo seguinte programa?

Resposta:

```
(7.000000 + 7.000000i)
(-3.000000 + -1.000000i)
(-2.000000 + 23.000000i)
(0.536585 + 0.170732i)
```

Operadores relacionais

Operação	Operador
Menor	<
Menor ou igual	<=
Maior	>
Maior ou igual	>=
Igual	==
Diferente	!=

Operadores relacionais

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  _Bool a, b, c, d;
  a = (2 > 3);
  b = ((2-7) >= -5);
  c = (b != b);
  d = (a == b);
  printf("%d %d %d %d\n", a, b, c, d);
  return 0;
}
```

Operadores relacionais

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  _Bool a, b, c, d;
  a = (2 > 3);
  b = ((2-7) >= -5);
  c = (b != b);
  d = (a == b);
  printf("%d %d %d %d\n", a, b, c, d);
  return 0;
}
```

Resposta: 0 1 0 0

Operadores relacionais

Comparação de valores de ponto flutuante

```
if (fabs(x - y) <= DBL_EPSILON) {
  /* x igual a y */
} else {
  /* x diferente de y */
}
```

Operadores lógicos

Operação	Operador
Complemento lógico (negação)	!
Conjunção e disjunção	&& e ||

Operadores lógicos

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  _Bool a, b, c, d;
  int e = 5;
  a = (2 > 3) && (++e <= 3);
  b = (1 == 1) || (++e <= 3);
  c = a && b;
  d = a || b;
  printf("%d %d %d %d %d\n",
                        a, b, c, d, e);
  return 0;
}
```

Operadores lógicos

Exemplo

O que é impresso pelo seguinte programa?

```
#include <stdio.h>
int main(void) {
  _Bool a, b, c, d;
  int e = 5;
  a = (2 > 3) && (++e <= 3);
  b = (1 == 1) || (++e <= 3);
  c = a && b;
  d = a || b;
  printf("%d %d %d %d %d\n",
                         a, b, c, d, e);
  return 0;
}
```

Resposta: 0 1 0 1 5

Operadores binários

<table>
<tr><th>Operação</th><th>Operador</th></tr>
<tr><td>Deslocamento binário</td><td><< e >></td></tr>
<tr><td>Operadores lógicos binários</td><td>&, |, ^ e ~</td></tr>
</table>

Operadores binários: deslocamento à esquerda

`a << b`

- Desloca `b` bits para a esquerda
- Preenche espaços à direita com zeros
- Equivale a: $a \times 2^b$
- Definida para `b` menor que o tamanho de `a` e tipos inteiros não sinalizados ou `a` não negativo.

Operadores binários: deslocamento à esquerda

Exemplo. O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  short int a, b, c;
  a = 64;
  b = a << 3;
  c = a << 6;
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

Operadores binários: deslocamento à esquerda

Exemplo. O que é impresso pelo programa ao lado?

Resposta: 64 512 4096

```
#include <stdio.h>
int main(void) {
  short int a, b, c;
  a = 64;
  b = a << 3;
  c = a << 6;
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

	Representação binária	Valor decimal
a	0000000001000000	64
a	00000000000000000000000001000000	64
a << 3	00000000000000000000001000000000	512
b	0000001000000000	512
a << 6	00000000000000000001000000000000	4.096
c	0001000000000000	4.096

Operadores binários: deslocamento à direita

`a >> b`

- Desloca `b` bits para a direita
- Preenche espaços à esquerda com zeros
- Equivale à divisão inteira: $a \div 2^b$
- Definida para `b` menor que o tamanho de `a` e para tipos inteiros não sinalizados ou `a` não negativo.

Operadores binários: deslocamento à direita

Exemplo. O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  short int a, b, c;
  a = 5000;
  b = a >> 3;
  c = a >> 6;
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

Operadores binários: deslocamento à direita

Exemplo. O que é impresso pelo programa ao lado?

Resposta: 5000 625 78

```
#include <stdio.h>
int main(void) {
  short int a, b, c;
  a = 5000;
  b = a >> 3;
  c = a >> 6;
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

	Representação binária	Valor decimal
a	0001001110001000	5.000
a	00000000000000000001001110001000	5.000
a >> 3	00000000000000000000001001110001	625
b	0000001001110001	625
a >> 6	00000000000000000000000001001110	78
c	0000000001001110	78

Operadores lógicos binários

Operação	Operador
Conjunção	&
Disjunção	|
Disjunção exclusiva	^
Negação	~

Operadores lógicos binários

Exemplo. O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  short int a = 123, b = 20;
  int c = a & b;
  int d = a | b;
  int e = a ^ b;
  int f = ~a;
  printf("%d %d\n", a, b);
  printf("%d %d %d %d\n", c, d, e, f);
  return 0;
}
```

Operadores lógicos binários

Exemplo. O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  short int a = 123, b = 20;
  int c = a & b;
  int d = a | b;
  int e = a ^ b;
  int f = ~a;
  printf("%d %d\n", a, b);
  printf("%d %d %d %d\n", c, d, e, f);
  return 0;
}
```

Resposta:

123 20

16 127 111 -124

	Representação binária	Valor decimal
a	0000000001111011	123
b	0000000000010100	20
c	00000000000000000000000000010000	16
d	00000000000000000000000001111111	127
e	00000000000000000000000001101111	111
f	11111111111111111111111110000100	−124

Operador condicional

$$\langle OpCond \rangle \quad ::= \quad \langle CondBool \rangle \ \textbf{?} \ \langle Expr \rangle \ \textbf{:} \ \langle ExprCond \rangle$$

$$\langle ExprCond \rangle \quad ::= \quad \langle OpCond \rangle \mid \langle Expr \rangle$$

Exemplo: `(a > 3) ?  x :  y`

Operador condicional

$$\langle \mathit{OpCond} \rangle \quad ::= \quad \langle \mathit{CondBool} \rangle \ \textbf{?} \ \langle \mathit{Expr} \rangle \ \textbf{:} \ \langle \mathit{ExprCond} \rangle$$

$$\langle \mathit{ExprCond} \rangle \quad ::= \quad \langle \mathit{OpCond} \rangle \mid \langle \mathit{Expr} \rangle$$

Exemplo: `(a > 3) ? x : y`

Exemplo. O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  int i = 20, j;
  j = i > 30 ? 2 + i : 2 * i;
  printf("%d\n", j);
  j = i < 30 ? 2 + i : 2 * i;
  printf("%d\n", j);
  return 0;
}
```

Operador condicional

$$\langle OpCond \rangle ::= \langle CondBool \rangle \; \textbf{?} \; \langle Expr \rangle \; \textbf{:} \; \langle ExprCond \rangle$$

$$\langle ExprCond \rangle ::= \langle OpCond \rangle \mid \langle Expr \rangle$$

Exemplo: `(a > 3) ? x : y`

Exemplo. O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  int i = 20, j;
  j = i > 30 ? 2 + i : 2 * i;
  printf("%d\n", j);
  j = i < 30 ? 2 + i : 2 * i;
  printf("%d\n", j);
  return 0;
}
```

Resposta:

40

22

Operador de atribuição

$\langle \mathit{OpAtrib} \rangle \quad ::= \langle \mathit{Variável} \rangle = \langle \mathit{Expr} \rangle$

A operação de atribuição simples é realizada em três passos:

- O operando esquerdo, $\langle \mathit{Variável} \rangle$, é avaliado para definir o endereço utilizado na atribuição.
- O operando direito, $\langle \mathit{Expr} \rangle$, é avaliado e o valor obtido é convertido na versão não qualificada do tipo do operando esquerdo.
- O resultado convertido é o valor resultante da operação, que é armazenado no endereço determinado pelo operando esquerdo.

A avaliação dos operandos pode ocorrer em qualquer ordem.

Atribuições compostas

⟨*OpAtribComposta*⟩ ::= ⟨*Variável*⟩ ⟨*Oper*⟩= ⟨*Expr*⟩

⟨*Oper*⟩ ::= ***** | **/** | **%** | **+** | **-** | **«** | **»** | **&** | **^** | **|**

Atribuições compostas

$\langle OpAtribComposta \rangle$ ::= $\langle Variável \rangle$ $\langle Oper \rangle$= $\langle Expr \rangle$

$\langle Oper \rangle$ ::= * | / | % | + | - | « | » | & | ^ | |

Exemplo

Atribuição composta	Expressão equivalente
`a *= b`	`a = a * (b)`
`prest %= n * juros / 100`	`prest = prest % (n * juros / 100)`
`valor1 += valor2 >>= saldo`	`valor1 = valor1 + (valor2 = valor2 >> (saldo))`

Atribuições compostas

⟨*OpAtribComposta*⟩ ::= ⟨*Variável*⟩ ⟨*Oper*⟩= ⟨*Expr*⟩

⟨*Oper*⟩ ::= * | / | % | + | - | « | » | & | ^ | |

Exemplo

Atribuição composta	Expressão equivalente
`a *= b`	`a = a * (b)`
`prest %= n * juros / 100`	`prest = prest % (n * juros / 100)`
`valor1 += valor2 >>= saldo`	`valor1 = valor1 + (valor2 = valor2 >> (saldo))`

Observação:

Nas atribuições compostas o operando esquerdo é avaliado uma única vez, enquanto que na atribuição simples equivalente ocorrem duas avaliações do mesmo termo.

Operador vírgula

$\langle \textit{OpVírgula} \rangle \quad ::= \ \langle \textit{ExprAtrib} \rangle \mid \langle \textit{OpVírgula} \rangle \ \mathbf{,} \ \langle \textit{ExprAtrib} \rangle$

A sequência de expressões, geralmente de atribuição, é avaliada da esquerda para a direita.

Operador vírgula

$\langle OpVírgula \rangle \quad ::= \ \langle ExprAtrib \rangle \mid \langle OpVírgula \rangle \ \textbf{,} \ \langle ExprAtrib \rangle$

A sequência de expressões, geralmente de atribuição, é avaliada da esquerda para a direita.

Exemplo

O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  int a = 2, b = 3, c = 1, d;
  b = (a = 4, c =  2 + a, 37 + c);
  d = (a + 10, 26);
  printf("%d %d %d %d\n", a, b, c, d);
  return 0;
}
```

Operador vírgula

$\langle \mathit{OpVírgula} \rangle \quad ::= \langle \mathit{ExprAtrib} \rangle \mid \langle \mathit{OpVírgula} \rangle \ , \ \langle \mathit{ExprAtrib} \rangle$

A sequência de expressões, geralmente de atribuição, é avaliada da esquerda para a direita.

Exemplo

O que é impresso pelo programa ao lado?

Resposta: 4 43 6 26

```
#include <stdio.h>
int main(void) {
  int a = 2, b = 3, c = 1, d;
  b = (a = 4, c =  2 + a, 37 + c);
  d = (a + 10, 26);
  printf("%d %d %d %d\n", a, b, c, d);
  return 0;
}
```

Operador de tamanho

⟨*OpTamanho*⟩ ::= **sizeof** ⟨*Expr*⟩ | **sizeof** **(** ⟨*Tipo*⟩ **)**

Normalmente o resultado é obtido sem avaliação do operando (que é avaliado apenas se é um vetor de tamanho variável).

Operador de conversão de tipo

$\langle$*OpConversão*$\rangle$ ::= **(** $\langle$*Tipo*$\rangle$ **)** $\langle$*Expr*$\rangle$

Exemplo

Algumas conversões válidas:

`(int)(3.2 * 3)`	Converte o valor 9,6 do tipo `double` no valor `9` do tipo `int`.
`(float)(12E2 + 4)`	Converte o valor 1204, 0 do tipo `double` no valor 1204, 0 do tipo `float`.
`(short int)taxa * 4.3F`	Converte o valor de `taxa` em um valor do tipo `short int`. O tipo da expressão continua sendo `float`.

Funções predefinidas

Algumas funções matemáticas declaradas no arquivo-cabeçalho `math.h`:

Função	Descrição
`double sin(double a)`	seno de `a`
`double cos(double a)`	cosseno de `a`
`double tan(double a)`	tangente de `a`
`double asin(double a)`	arco seno de `a`
`double acos(double a)`	arco cosseno de `a`
`double atan(double a)`	arco tangente de `a`
`double exp(double a)`	valor de e^{a}
`double log(double a)`	logaritmo natural de `a`
`double log2(double a)`	logaritmo base 2 de `a`
`double sqrt(double a)`	raiz quadrada de `a`
`double pow(double a, double b)`	valor de a^{b}
`double fabs(double a)`	valor absoluto de `a`
`double round(double a)`	valor arredondado de `a`

Em geral, para cada função `fun` existem versões `funl` e `funf`.

Definição de tipos

⟨*DefTipo*⟩ ::= **typedef** ⟨*Tipo*⟩ ⟨*TpIdent*⟩ {, ⟨*TpIdent*⟩ } ;

Uma declaração `typedef` não cria novos tipos. Apenas torna ⟨*TpIdent*⟩ um sinônimo para o tipo ⟨*Tipo*⟩.

Definição de tipos

Exemplo

As declarações

```
typedef const int int_tp;
typedef int int32_t, *int32_ptr;
typedef short f_tp(int), n_tp;
typedef int a_tp[x], b_tp[][4];
```

Fazem com que

`const int`	≡	`int_tp`	`int`	≡	`int32_t`
`int *`	≡	`int32_ptr`	`short (int)`	≡	`f_tp`
`short`	≡	`n_tp`	`int [x]`	≡	`a_tp`
`int [][4]`	≡	`b_tp`			

Definições disponíveis

O cabeçalho `stdint.h` contém as seguintes definições:

sinalizado	não sinalizado
`int8_t`	`uint8_t`
`int16_t`	`uint16_t`
`int32_t`	`uint32_t`
`int64_t`	`uint64_t`

Ordem de avaliação

Associatividade	Precedência	Operador
E	0	() (chamada a função), [] (indexação), -> (seleção indireta), . (seleção direta)
E	1	++ (incremento pós), -- (decremento pós)
D	2	++ (incremento pré), -- (decremento pré)
D	3	sizeof (tamanho), & (endereço), * (acesso indireto), - (menos unário), + (mais unário), ! (negação lógica), ~ (negação binária)
D	4	(⟨*tipo*⟩) (conversão de tipo)
E	5	* (multiplicação), / (divisão), % (mod)
E	6	+ (adição), - (subtração)
E	7	<<, >> (deslocamentos)

Ordem de avaliação

Associa-tividade	Prece-dência	Operador
E	8	`<`, `>`, `<=` e `>=` (relacionais)
E	9	`==` (igualdade), `!=` (desigualdade)
E	10	`&` (conjunção binária)
E	11	`^` (disjunção exclusiva binária)
E	12	`|` (disjunção binária)
E	13	`&&` (conjunção lógica)
E	14	`||` (disjunção lógica)
D	15	`? :` (condicional)
D	16	`=` (atribuição), `op=` (atribuição composta)
E	17	`,` (vírgula)

Ordem de avaliação

Exemplo

Coloque as expressões na forma parentética equivalente.

Expressão original	Forma parentética equivalente
`2 * a - a % 3 / 4`	
`b + c / b - c / d`	
`a + b + c - - d + +3 / -5 - -e * f`	

Ordem de avaliação

Exemplo

Coloque as expressões na forma parentética equivalente.

Expressão original	Forma parentética equivalente
`2 * a - a % 3 / 4`	`(2 * a) - ((a % 3) / 4)`
`b + c / b - c / d`	
`a + b + c - - d + +3 / -5 - -e * f`	

Ordem de avaliação

Exemplo

Coloque as expressões na forma parentética equivalente.

Expressão original	Forma parentética equivalente
`2 * a - a % 3 / 4`	`(2 * a) - ((a % 3) / 4)`
`b + c / b - c / d`	`(b + (c / b)) - (c / d)`
`a + b + c - - d + +3 / -5 - -e * f`	

Ordem de avaliação

Exemplo

Coloque as expressões na forma parentética equivalente.

Expressão original	Forma parentética equivalente
`2 * a - a % 3 / 4`	`(2 * a) - ((a % 3) / 4)`
`b + c / b - c / d`	`(b + (c / b)) - (c / d)`
`a + b + c - - d + +3 / -5 - -e * f`	`((((a + b) + c) - (- d)) + ((+3) / -5)) - ((-e) * f)`

Ordem de avaliação - sequenciamento

Em geral a avaliação de uma expressão não é sequenciada.
Os seguintes pontos de sequenciamento garantem que todas as operações já iniciadas serão completadas antes das próximas avaliações:

Término da expressão

```
b = a + a++;
```

Operadores lógicos

```
(a++ > b) && (a < d)
```

Condicional

```
(a++ > b--) ?  a :  b
```

Ordem de avaliação - sequenciamento

Os seguintes pontos de sequenciamento garantem que todas as operações já iniciadas serão completadas antes das próximas avaliações:

Vírgula

```
a = ++b, b * c
```

Chamada a função

```
fun(x + y, ++x)
```

O tipo das operações

O seguinte procedimento é aplicado a todos os operadores de uma expressão:

- Cada operando é convertido em um tipo real comum, sem mudança do domínio: real ou complexo.
- O tipo do resultado é o tipo comum obtido. O domínio do resultado será
    - complexo, se um dos operandos pertencer ao domínio complexo, ou
    - real, se ambos pertencerem ao domínio real.

O tipo das operações

Determinação do tipo real comum

1) Se o tipo real correspondente a um dos operandos é `long double`, o outro operando é convertido em um tipo cujo tipo real correspondente é `long double`; senão
2) se tipo real correspondente a um dos operandos é `double`, o outro operando é convertido em um tipo cujo tipo real correspondente é `double`; senão
3) se tipo real correspondente a um dos operandos é `float`, o outro operando é convertido em um tipo cujo tipo real correspondente é `float`; senão

O tipo das operações

Determinação do tipo real comum

4) aplica-se a *promoção inteira* a ambos os operandos. Após a promoção inteira, prossegue-se com a determinação do tipo real comum:
   - 4.1) Se o tipo de ambos é igual, esse é o tipo real comum; senão
   - 4.2) se ambos são de tipos inteiros sinalizados, ou ambos são de tipos inteiros não sinalizados, o de menor ordem é convertido no tipo de maior ordem; senão
   - 4.3) se a ordem do tipo inteiro não sinalizado é maior ou igual à ordem do outro operando, então o operando com tipo inteiro sinalizado é convertido no tipo não sinalizado; senão
   - 4.4) se o tipo do operando com tipo inteiro sinalizado pode representar todos os valores do tipo inteiro não sinalizado, então o operando com o tipo não sinalizado é convertido no tipo sinalizado; senão
   - 4.5) ambos os operandos são convertidos no tipo inteiro não sinalizado correspondente ao tipo inteiro sinalizado.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Estruturas condicionais

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Comando if

⟨*ComandoIf*⟩ ::= **if (** ⟨*Condição*⟩ **)** ⟨*CláusulaEntão*⟩ [**else** ⟨*CláusulaSenão*⟩]

⟨*Condição*⟩ ::= Expressão do tipo escalar resultando em um valor verdadeiro (diferente de 0) ou falso (igual a 0).

⟨*CláusulaEntão*⟩ ::= ⟨*BlocoInstr*⟩

⟨*CláusulaSenão*⟩ ::= ⟨*BlocoInstr*⟩

⟨*BlocoInstr*⟩ ::= ⟨*Bloco*⟩ | ⟨*Instrução*⟩

⟨*Bloco*⟩ ::= **{** {⟨*Instrução*⟩} **}**

⟨*Instrução*⟩ ::= ⟨*DeclVarLocal*⟩ | ⟨*Comando*⟩

Comando if — sem cláusula-senão

if (⟨*Condição*⟩ **)** ⟨*CláusulaEntão*⟩

Exemplo

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30) {
    printf("%d maior que 30\n", a);
    a = a - 30;
  }
  printf("%d menor ou igual a 30\n", a);
  return 0;
}
```

Comando if — sem cláusula-senão

if (⟨*Condição*⟩ **)** ⟨*CláusulaEntão*⟩

Exemplo

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30) {
    printf("%d maior que 30\n", a);
    a = a - 30;
  }
  printf("%d menor ou igual a 30\n", a);
  return 0;
}
```

O que é impresso pelo programa ao lado, se for lido o número 38?

Comando if — sem cláusula-senão

if (⟨*Condição*⟩) ⟨*CláusulaEntão*⟩

Exemplo

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30) {
    printf("%d maior que 30\n", a);
    a = a - 30;
  }
  printf("%d menor ou igual a 30\n", a);
  return 0;
}
```

O que é impresso pelo programa ao lado, se for lido o número 38?

Resposta:

```
38 maior que 30
8 menor ou igual a 30
```

Comando if — sem cláusula-senão

Exemplo

O que é impresso pelo programa ao lado, se for lido o número 38?

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30)
    printf("%d maior que 30\n", a);
  printf("%d menor ou igual a 30\n", a);
  return 0;
}
```

Comando if — sem cláusula-senão

Exemplo

O que é impresso pelo programa ao lado, se for lido o número 38?

Resposta:

```
38 maior que 30
38 menor ou igual a 30
```

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30)
    printf("%d maior que 30\n", a);
  printf("%d menor ou igual a 30\n", a);
  return 0;
}
```

Comando if — com cláusula-senão

if (⟨*Condição*⟩ **)** ⟨*CláusulaEntão*⟩ **else** ⟨*CláusulaSenão*⟩

Exemplo

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30)
    printf("maior que 30\n");
  else
    printf("menor ou igual a 30\n");
  printf("fim");
  return 0;
}
```

Comando if — com cláusula-senão

if (⟨*Condição*⟩ **)** ⟨*CláusulaEntão*⟩ **else** ⟨*ClásulaSenão*⟩

Exemplo

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30)
    printf("maior que 30\n");
  else
    printf("menor ou igual a 30\n");
  printf("fim");
  return 0;
}
```

O que é impresso pelo programa ao lado, se o número lido for

a) maior que 30?
b) menor ou igual a 30?

Comando if — com cláusula-senão

if (⟨*Condição*⟩ **)** ⟨*CláusulaEntão*⟩ **else** ⟨*CláusulaSenão*⟩

Exemplo

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  if (a > 30)
    printf("maior que 30\n");
  else
    printf("menor ou igual a 30\n");
  printf("fim");
  return 0;
}
```

O que é impresso pelo programa ao lado, se o número lido for

a) maior que 30?
b) menor ou igual a 30?

Resposta:

a é maior que 30	*a é menor ou igual a 30*
`maior que 30` `fim`	`menor ou igual a 30` `fim`

Comandos if aninhados

Exemplo

```
#include <stdio.h>
int main(void) {
  int num, val, taxa;
  scanf("%d", &num);
  scanf("%d", &val);
  scanf("%d", &taxa);
  if (num > val) {
    printf(" 1 ");
    printf(" 2 ");
  } else {
    if (val > taxa)
      printf(" 3 ");
    printf(" 4 ");
  }
  printf(" 5\n");
  return 0;
}
```

Comandos if aninhados

Exemplo

```
#include <stdio.h>
int main(void) {
  int num, val, taxa;
  scanf("%d", &num);
  scanf("%d", &val);
  scanf("%d", &taxa);
  if (num > val) {
    printf(" 1 ");
    printf(" 2 ");
  } else {
    if (val > taxa)
      printf(" 3 ");
    printf(" 4 ");
  }
  printf(" 5\n");
  return 0;
}
```

O que é impresso pelo programa ao lado, se

a) `num` for maior que `val`?
b) `num` for menor ou igual a `val` e `val` for maior que `taxa`?
c) `num` for menor ou igual a `val` e `val` for menor ou igual a `taxa`?

Comandos if aninhados

Exemplo

```
#include <stdio.h>
int main(void) {
  int num, val, taxa;
  scanf("%d", &num);
  scanf("%d", &val);
  scanf("%d", &taxa);
  if (num > val) {
    printf(" 1 ");
    printf(" 2 ");
  } else {
    if (val > taxa)
      printf(" 3 ");
    printf(" 4 ");
  }
  printf(" 5\n");
  return 0;
}
```

O que é impresso pelo programa ao lado, se

a) num for maior que val?
b) num for menor ou igual a val e val for maior que taxa?
c) num for menor ou igual a val e val for menor ou igual a taxa?

Resposta: a) 1 2 5
b) 3 4 5
c) 4 5

Comando if — cláusulas vazias

```
if (a > 5) { }

if (a > 23) ;

if (a < 12) { }
else ;
```

O uso de cláusulas vazias pode ser justificado para melhorar a legibilidade, tornando explícito que nada deve ser executado em certas situações.

Comando switch

$\langle \mathit{ComandoSwitch} \rangle$ ::= **switch (** $\langle \mathit{ExprInt} \rangle$ **)** $\langle \mathit{CorpoSwitch} \rangle$

$\langle \mathit{CorpoSwitch} \rangle$ ::= $\langle \mathit{BlocoInstr} \rangle$ | $\langle \mathit{CláusulaSwitch} \rangle$ | **{** $\{\langle \mathit{CláusulaSwitch} \rangle\}$ **}**

$\langle \mathit{CláusulaSwitch} \rangle$ ::= $\langle \mathit{RótuloSwitch} \rangle$ { $\langle \mathit{BlocoInstr} \rangle$ }

$\langle \mathit{RótuloSwitch} \rangle$ ::= **case** $\langle \mathit{ExprCteInt} \rangle$ **:** | **default :**

$\langle \mathit{ExprInt} \rangle$ ::= Expressão de um tipo inteiro.

$\langle \mathit{ExprCteInt} \rangle$ ::= Expressão constante de um tipo inteiro.

Comando switch

```
switch (a * 2) {
  case 14:
  case 8:
    x = 3 - a;
  case 4:
    x = a;
  default:
    x = 0;
}
```

- Avaliação
- Comparação
- Transferência
- Finalização

Comando switch

```
⇒   switch (a * 2) {
      case 14:
      case 8:
        x = 3 - a;
      case 4:
        x = a;
      default:
        x = 0;
    }
```

- Avaliação
  Avalia a expressão.
- Comparação
- Transferência
- Finalização

Comando switch

```
      switch (a * 2) {
⇒       case 14:
⇒       case 8:
          x = 3 - a;
⇒       case 4:
          x = a;
⇒       default:
          x = 0;
      }
```

- Avaliação
- Comparação
  Compara o resultado com os rótulos das cláusulas.
- Transferência
- Finalização

Comando switch

```
        switch (a * 2) {
          case 14:
          case 8:
⇒           x = 3 - a;
          case 4:
            x = a;
          default:
            x = 0;
        }
```

- Avaliação
- Comparação
- Transferência
  Transfere o controle para o comando da primeira cláusula com rótulo igual ao da expressão, ou para a cláusula default.
- Finalização

Comando switch

```
switch (a * 2) {
  case 14:
  case 8:
    x = 3 - a;
  case 4:
    x = a;
  default:
    x = 0;
}
```

⇒

- Avaliação
- Comparação
- Transferência
- Finalização
  Prossegue com o próximo comando.

Comando switch

Exemplo.

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  switch (2 + a) {
    case 23: printf("primeiro\n");
    default: printf("nenhum\n");
    case 5 * 9 / 3:
    case 2:  printf("segundo\n");
    case 4:  printf("ultimo\n");
  }
  printf("fim\n");
  return 0;
}
```

Comando switch

Exemplo.

O que é impresso pelo programa ao lado, se o número lido for

a) 21?
b) 13?
c) 10?

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  switch (2 + a) {
    case 23: printf("primeiro\n");
    default: printf("nenhum\n");
    case 5 * 9 / 3:
    case 2:  printf("segundo\n");
    case 4:  printf("ultimo\n");
  }
  printf("fim\n");
  return 0;
}
```

Comando switch

Exemplo.

O que é impresso pelo programa ao lado, se o número lido for

a) 21?
b) 13?
c) 10?

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  switch (2 + a) {
    case 23: printf("primeiro\n");
    default: printf("nenhum\n");
    case 5 * 9 / 3:
    case 2:  printf("segundo\n");
    case 4:  printf("ultimo\n");
  }
  printf("fim\n");
  return 0;
}
```

Resposta:

$a = 21$	$a = 13$	$a = 10$
primeiro	segundo	nenhum
nenhum	ultimo	segundo
segundo	fim	ultimo
ultimo		fim
fim		

Comando switch - break

O comando `break` interrompe a execução do `switch` que o contém.

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  switch (2 + a) {
    case 23: printf("primeiro apos 23\n");
             break;
    default:
    case 15:
    case 2:  printf("primeiro apos 2\n");
             break;
    case 4:  printf("primeiro apos 4\n");
             break;
  }
  printf("fim\n");
  return 0;
}
```

Comando switch - break

O comando `break` interrompe a execução do `switch` que o contém.

O programa ao lado, se o valor lido for 21, imprimirá:

```
primeiro apos 23
fim
```

```
#include <stdio.h>
int main(void) {
  int a;
  scanf("%d", &a);
  switch (2 + a) {
    case 23: printf("primeiro apos 23\n");
             break;
    default:
    case 15:
    case 2:  printf("primeiro apos 2\n");
             break;
    case 4:  printf("primeiro apos 4\n");
             break;
  }
  printf("fim\n");
  return 0;
}
```

Comandos switch aninhados

O que é impresso pelo comando comando switch ao lado se o valor de a for igual a 2?

```
switch (a) {
  case 1:
    printf("rotulo 1\n");
    break;
  case 2:
    a = 3 * a;
    switch (a) {
      case 6:
        printf("rotulo 6, apos 2\n");
        break;
      case 8:
        printf("rotulo 8, apos 2\n");
    }
    printf("rotulo 2\n");
    break;
  default:
    printf("rotulo default\n");
}
```

Comandos switch aninhados

O que é impresso pelo comando comando `switch` ao lado se o valor de a for igual a 2?

Resposta:

```
rotulo 6, apos 2
rotulo 2
```

```
switch (a) {
  case 1:
    printf("rotulo 1\n");
    break;
  case 2:
    a = 3 * a;
    switch (a) {
      case 6:
        printf("rotulo 6, apos 2\n");
        break;
      case 8:
        printf("rotulo 8, apos 2\n");
    }
    printf("rotulo 2\n");
    break;
  default:
    printf("rotulo default\n");
}
```

Comando switch — situações especiais

As sguintes situações especiais devem ser evitadas:

- Comportamento invariável, devido a cláusulas vazias ou comando contendo apenas a cláusula `default`.
- Cláusula fora de bloco, pois dificulta a legibilidade.
- Declarações no corpo de um `switch`, pois podem não ser iniciadas adequadamente.
- Declarações de vetores variáveis. Todo o `switch` deve estar no escopo do vetor.

Obrigações de prova

Ao usar comandos condicionais deve-se assegurar que

1. A condição do comando `if` pode assumir tanto o valor verdadeiro quanto o falso.
2. Todas as alternativas devem poder ser executadas.

Promovendo a manutenibilidade

As condições de um comando condicional devem ser mantidas simples:

- Modificando os operadores.
  `!(a < c)` é equivalente a `(a >= c)`.
- Modificando a estrutura para eliminar (ou introduzir) disjunções e conjunções.
  Uma disjunção pode ser implementada com uma cláusula-senão e uma conjunção com comandos aninhados.
- Evitando operadores com efeitos colaterais.

Deve-se também evitar o uso de `else` ambíguo.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Estruturas de repetição

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Comando while

$\langle$*ComandoWhile*$\rangle$ ::= **while (** $\langle$*Condição*$\rangle$ **)** $\langle$*CláusulaRepetição*$\rangle$

$\langle$*Condição*$\rangle$ ::= Expressão do tipo escalar resultando em um valor verdadeiro (diferente de 0) ou falso (igual a 0).

$\langle$*ClásulaRepetição*$\rangle$ ::= $\langle$*BlocoInstr*$\rangle$

Comando while

Exemplo

O que é impresso pelo programa ao lado?

```
#include <stdio.h>
int main(void) {
  int qtd = 1;
  while (qtd <= 1500) {
    printf("%f\n", 1.0/(qtd + 1));
    qtd = qtd + 1;
  }
  printf("fim\n");
  return 0;
}
```

Comando while

Exemplo

O que é impresso pelo programa ao lado?

Resposta: os 1.500 primeiros termos da sequência $\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots$.

```
#include <stdio.h>
int main(void) {
  int qtd = 1;
  while (qtd <= 1500) {
    printf("%f\n", 1.0/(qtd + 1));
    qtd = qtd + 1;
  }
  printf("fim\n");
  return 0;
}
```

Comando while

Exemplo

O programa ao lado lê um número assegurando que o número lido esteja na faixa $[1, 229]$

```
#include <stdio.h>
int main(void) {
  int num = -1;
  while ((num <= 0) || (num >= 230)) {
    printf("Digite 0 < numero < 230: ");
    scanf("%d", &num);
  }
  printf("2 x %d = %d\n", num, (2 * num));
  printf("fim\n");
  return 0;
}
```

Comando while

Exemplo

O programa ao lado lê um número assegurando que o número lido esteja na faixa $[1, 229]$

```
#include <stdio.h>
int main(void) {
  int num = -1;
  while ((num <= 0) || (num >= 230)) {
    printf("Digite 0 < numero < 230: ");
    scanf("%d", &num);
  }
  printf("2 x %d = %d\n", num, (2 * num));
  printf("fim\n");
  return 0;
}
```

- Pode-se realizar uma leitura inicial em vez de atribuir o valor -1 à variável `num`.
- Porém, o melhor é usar uma estrutura de repetição mais adequada, que garanta pelo menos uma repetição.

Comando while

Exemplo

O programa ao lado lê um número *N* e a seguir lê *N* números pares. Se *N* for negativo ou zero o programa termina sem realizar nenhuma leitura adicional.

```
#include <stdio.h>
int main(void) {
  int n, x, i = 0;
  printf("Digite a qtd. de numeros: ");
  scanf("%d", &n);
  printf("Digite %d numeros pares\n", n);
  while (i < n) {
    x = 1;
    while ((x % 2) != 0) {
      printf("Numero %d: ", i + 1);
      scanf("%d", &x);
    }
    i++;
  }
  return 0;
}
```

Comando do

$\langle$*ComandoDo*$\rangle$ ::= **do** $\langle$*CláusulaRepetição*$\rangle$ **while (** $\langle$*Condição*$\rangle$ **) ;**

$\langle$*Condição*$\rangle$ ::= Expressão do tipo escalar resultando em um valor verdadeiro (diferente de 0) ou falso (igual a 0).

$\langle$*CláusulaRepetição*$\rangle$ ::= $\langle$*BlocoInstr*$\rangle$

Comando do

Exemplo

O programa ao lado lê um número assegurando que o número lido esteja na faixa $[1, 229]$

```
#include <stdio.h>
int main(void) {
  int num;
  do {
    printf("Digite 0 < numero < 230: ");
    scanf("%d", &num);
  } while ((num <= 0) || (num >= 230));
  printf("2 x %d = %d\n", num, 2 * num);
  printf("fim\n");
  return 0;
}
```

Comandos do e while

Exemplo

O programa ao lado lê um número positivo N e imprime o fatorial de N.

```
#include <stdio.h>
int main(void) {
  int num;
  long long int fat = 1LL;
  do {
    printf("Digite um numero >= 0: ");
    scanf("%d", &num);
  } while (num < 0);
  while (num > 1)
    fat = fat * num--;
  printf("Fatorial= %lld\n", fat);
  return 0;
}
```

Comandos do e while

Exemplo

O programa ao lado lê um número positivo N e imprime o fatorial de N.

```
#include <stdio.h>
int main(void) {
  int num;
  long long int fat = 1LL;
  do {
    printf("Digite um numero >= 0: ");
    scanf("%d", &num);
  } while (num < 0);
  while (num > 1)
    fat = fat * num--;
  printf("Fatorial= %lld\n", fat);
  return 0;
}
```

Observação: se `long long int` for implementado com 64 bits, os fatoriais não são calculados corretamente para $N \geq 21$.

Comando for

⟨*ComandoFor*⟩ ::= **for (** [⟨*IniFor*⟩] ; [⟨*Condição*⟩] ; [⟨*FimIter*⟩] **)** ⟨*CláusulaRepetição*⟩

⟨*IniFor*⟩ ::= ⟨*ListaExprC*⟩ | ⟨*DeclVarLocal*⟩

⟨*FimIter*⟩ ::= ⟨*ListaExprC*⟩

⟨*Condição*⟩ ::= Expressão do tipo escalar resultando em um valor verdadeiro (diferente de 0) ou falso (igual a 0).

⟨*ListaExprC*⟩ ::= ⟨*ExprC*⟩ | ⟨*ListaExprC*⟩ , ⟨*ExprC*⟩

⟨*ExprC*⟩ ::= Expressão consistindo de operadores e operandos.

⟨*DeclVarLocal*⟩ ::= Declaração de variáveis locais.

⟨*CláusulaRepetição*⟩ ::= ⟨*BlocoInstr*⟩

Execução do comando for

Início do for. Os comandos e declarações na cláusula inicial são executados uma única vez.

Teste da condição. A condição é avaliada.

Iteração. Se a condição for verdadeira, os comandos da cláusula de repetição são executados.

Término da iteração. Os comandos da cláusula final são executados, após o que o controle é transferido para uma nova avaliação da condição, reiniciando o processo.

Comando for

Exemplo

O programa ao lado lê imprime a soma dos 200 primeiros números naturais.

```
#include <stdio.h>
int main(void) {
  int soma = 0;
  for (int i = 1; i <= 200; i++)
    soma = soma + i;
  printf("soma: %d\n", soma);
  return 0;
}
```

Comando for

Exemplo

O programa ao lado lê imprime a soma dos 200 primeiros números naturais.

```
#include <stdio.h>
int main(void) {
  int soma = 0;
  for (int i = 1; i <= 200; i++)
    soma = soma + i;
  printf("soma: %d\n", soma);
  return 0;
}
```

Observação: a declaração de variáveis na cláusula inicial de um comando `for` é própria do padrão ISO/IEC 9899:1999.

Comando for

Exemplo

O programa ao lado ilustra o uso de chamadas a função nas cláusulas inicial e final de um comando for.

```
#include <stdio.h>
int main(void) {
  int num;
  long fat;
  printf("Digite um numero >= 0: ");
  for (scanf("%d", &num) ;
       num < 0 ;
       printf("Digite um numero >= 0: "),
       scanf("%d", &num))
  { }
  for (fat = 1L; num > 1; num--)
    fat = fat * num;
  printf("Fatorial = %ld\n", fat);
  return 0;
}
```

Iterações infinitas e cláusulas vazias

Os comandos `do` e `while` podem ter cláusulas vazias.

```
while (a > 5) { }          do { } while (a > 5) ;

while (a > 5) ;            do ; while (a > 5) ;
```

Tanto a cláusula de repetição quanto as cláusulas inicial e final do comando `for` podem ser vazias.

```
for ( ; ; ) { }            for ( ; ; ) ;
```

Comando break

O comando `break` interrompe a iteração que o contém, encerrando o comando de iteração.

Comando break

O comando `break` interrompe a iteração que o contém, encerrando o comando de iteração.

Exemplo

```
for (p = 1; p <= n; p++) {
  if (soma + p > lim) {
    break;
  }
  soma = soma + p;
}
```

```
while (p <= n) {
  if (soma + p > lim) {
    break;
  }
  soma = soma + p++;
}
```

Comando break

Exemplo

O que é impresso pelo programa ao lado se o número lido for igual a 4?

```
#include <stdio.h>
#include <stdbool.h>
int main(void) {
  int n, b, soma;
  scanf("%d", &n);
  while (n > 0) {
    soma = 1; b = 1;
    printf("%d", b++);
    while (true) {
      if (b > n)
        break;
      printf(" + %d", b);
      soma = soma + b++;
    }
    printf(" = %d\n", soma);
    n--;
  }
  return 0;
}
```

Comando break

Exemplo

O que é impresso pelo programa ao lado se o número lido for igual a 4?

```
#include <stdio.h>
#include <stdbool.h>
int main(void) {
  int n, b, soma;
  scanf("%d", &n);
  while (n > 0) {
    soma = 1; b = 1;
    printf("%d", b++);
    while (true) {
      if (b > n)
        break;
      printf(" + %d", b);
      soma = soma + b++;
    }
    printf(" = %d\n", soma);
    n--;
  }
  return 0;
}
```

Resposta:

1 + 2 + 3 + 4 = 10

1 + 2 + 3 = 6

1 + 2 = 3

1 = 1

Comando continue

O comando `continue` interrompe a iteração que o contém, reiniciando o comando iterativo,

- a partir de uma nova avaliação da condição (para os comandos `do` e `while`) ou
- a partir da cláusula de fim de iteração (para o comando `for`).

Comando continue

Exemplo

O programa ao lado lê seis números pares e maiores do que 0, imprimindo para cada número lido N a soma dos naturais de 1 a N.

O comando continue é usado para reiniciar o processamento caso o número lido não seja válido.

```
#include <stdio.h>
int main(void) {
  int num, soma, qtd=0;
  while (qtd < 6) {
    printf("Digite um numero par > 0: ");
    scanf("%d", &num);
    if ((num % 2) != 0 || (num <= 0))
      continue;
    qtd++; soma = 0;
    for (int i = 1; i <= num; i++)
      soma = soma + i;
    printf("x = %d, s(x) = %d\n",
                              num, soma);
  }
  printf("fim\n");
  return 0;
}
```

Comando goto

O comando `goto` provoca o desvio incondicional do fluxo da execução para o comando rotulado por seu rótulo.

- O rótulo de um `goto` deve estar no escopo da função que o contém.

Comando goto

Exemplo

A função ao lado imprime os números naturais até o primeiro maior ou igual a `lim`. A impressão ocorre

- de 1 em 1, se `modo` for igual a 1;
- de 2 em 2, se `modo` for igual a 2; ou
- de 3 em 3, se `modo` for igual a 3.

```
void fun(int modo, int lim) {
  int num = 0;
  do {
    if (modo == 1) {
      goto s1;
    } else {
      if (modo == 2) {
        goto s2;
      }
    }
    num++;
s2: num++;
s1: num++;
    printf("%d ", num);
  } while (num < lim);
}
```

Programação sem goto

- O comando `goto` não é compatível com a programação estruturada.
- Todo programa pode ser desenvolvido usando-se para interromper o fluxo sequencial da execução apenas as estruturas convencionais de decisão e repetição.
- O uso do `goto` torna os programas mais difíceis de ser entendidos e modificados.

Outras formas de desvio e interrupção

- Para interrupção da execução de funções:
  - `return`
  - `exit`
  - `quick_exit` (função definida na versão 2011 do padrão da linguagem).
  - `_Exit`
  - `abort`
- Para desvios não locais
  - `setjmp`
  - `longjmp`

Obrigações de prova

O uso de comandos iterativos exige duas obrigações de prova:

1. Haverá pelo menos uma iteração.
2. As iterações eventualmente param.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

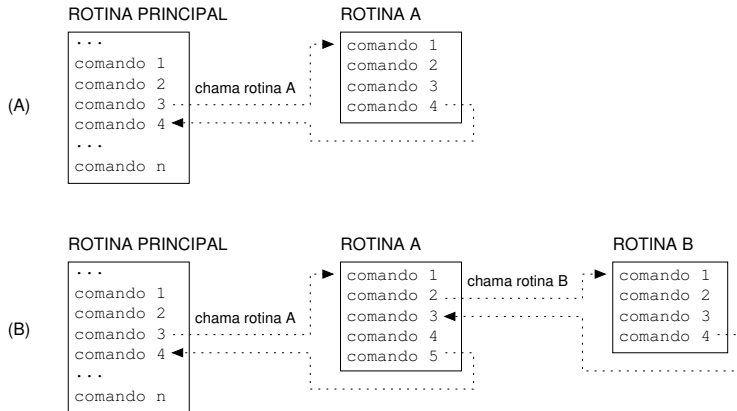
Elementos de programação em C

Funções e procedimentos

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Fluxo de execução

Funções e procedimentos

Função. Caracterizada por retornar um valor como resultado do processamento.

Procedimento. Caracterizado por não produzir valor de retorno.

Sintaxe da declaração

⟨*DeclFunção*⟩ ::= ⟨*Cabeçalho*⟩ **;**

⟨*DefiniçãoFunção*⟩ ::= ⟨*Cabeçalho*⟩ ⟨*CorpoFunção*⟩

⟨*Cabeçalho*⟩ ::= [**inline**] [**_Noreturn**] ⟨*DeclTipo*⟩ ⟨*IdentFunção*⟩ **(** ⟨*ListaParâmetros*⟩ [**, ...**] **)**

⟨*Cabeçalho*⟩ ::= [**inline**] [**_Noreturn**] ⟨*DeclTipo*⟩ ⟨*IdentFunção*⟩ **( )**

⟨*IdentFunção*⟩ ::= Identificador da função.

⟨*ListaParâmetros*⟩ ::= ⟨*Parâmetro*⟩ [**,** ⟨*ListaParâmetros*⟩]

⟨*CorpoFunção*⟩ ::= Declarações e comandos, entre chaves, que implementam a função.

Declaração, definição e protótipo

Declaração de função especifica o tipo do valor de retorno, a identificação da função e, opcionalmente, o tipo dos seus parâmetros.

Definição de função é a declaração que especifica o corpo da função, isto é, causa alocação de memória.

Protótipo de função é a declaração que contém o tipo dos parâmetros da lista de parâmetros ou `void`, se a função não possuir lista de parâmetros.

Declaração, definição e protótipo

Declaração de função especifica o tipo do valor de retorno, a identificação da função e, opcionalmente, o tipo dos seus parâmetros.

Definição de função é a declaração que especifica o corpo da função, isto é, causa alocação de memória.

Protótipo de função é a declaração que contém o tipo dos parâmetros da lista de parâmetros ou `void`, se a função não possuir lista de parâmetros.

Observação:

A definição de uma função induz sua declaração (consistindo do cabeçalho da função.

Declaração, definição e protótipo

Código	Classificação
`int funA();`	Declaração.
`int funB(void);`	Protótipo.
`void funC(int, float);`	Protótipo.
`void funD(int a, float b);`	Protótipo.
`float funE(int a) { /* codigo omitido */ }`	Definição. Declaração induzida: `float funE(int a)`

Declaração, definição e protótipo

Exemplo

No trecho de código ao lado, que funções estão definidas, apenas declaradas e declaradas como protótipos?

```
#include <stdio.h>
const int funB();
char *funC(int, float);
void funD(long int a);
int main(void) {
  float funE(void);
  /* codigo omitido */
  return 0;
}
long funA(int a) {
  /* codigo omitido */
}
```

Declaração, definição e protótipo

Exemplo

No trecho de código ao lado, que funções estão definidas, apenas declaradas e declaradas como protótipos?

Resposta:
definidas: main, funA
declaradas: funB
protótipos: funC, funD, funE

```
#include <stdio.h>
const int funB();
char *funC(int, float);
void funD(long int a);
int main(void) {
  float funE(void);
  /* codigo omitido */
  return 0;
}
long funA(int a) {
  /* codigo omitido */
}
```

Escopo e declaração implícita

Escopo de uma declaração de função:

- Bloco
- Arquivo

Entretanto, a definição de uma função não pode ter escopo de bloco.

Escopo e declaração implícita

Quando uma chamada a uma função `fun` ocorre fora do escopo de sua declaração, o compilador assume a existência de uma

Declaração implícita

```
int fun();
```

A declaração de uma função com a lista de parâmetros vazia faz com que o compilador não verifique a consistência entre os argumentos usados na chamada e os parâmetros declarados na definição da função.

Escopo e declaração implícita

Exemplo

O escopo do protótipo para a função `funA` vai do ponto de sua declaração até o fim da função `main`.

As referências a `funA` fora desse escopo são interpretadas no escopo de uma declaração implícita `int funA()`.

```
#include <stdio.h>
int main(void) {
  funA(4);
  int funA(double, int);
  funA(3.7, 66);
  funB("exem");
  return 0;
}
void funB(int a) {
  funA('a', 57);
}
```

A função `funB` possui uma declaração explícita (induzida por sua definição) e uma implícita.

Parâmetros e argumentos

Parâmetros são as variáveis declaradas na definição de uma função.

Argumentos são as expressões usadas na chamada a uma função.

Passagem de argumentos

Por valor. Uma cópia do argumento é atribuída ao parâmetro correspondente.

Por referência. O parâmetro passa a ser uma referência ao argumento usado na chamada.

Parâmetros e argumentos

Parâmetros são as variáveis declaradas na definição de uma função.

Argumentos são as expressões usadas na chamada a uma função.

Passagem de argumentos

Por valor. Uma cópia do argumento é atribuída ao parâmetro correspondente.

Por referência. O parâmetro passa a ser uma referência ao argumento usado na chamada.

Observação:

Em C, toda passagem de argumento é por valor!

Parâmetros e argumentos — restrições

- Parâmetros não podem conter especificador de classe, exceto `register`.
- Parâmetros não podem ser iniciados.
- Nas declarações, além dos tipos completos, os parâmetros podem ser de um tipo incompleto, ou de um tipo vetor de tamanho variável não especificado (`[*]`).
- Nas definições, os parâmetros só podem ser de um tipo completo, ou de um tipo vetor de tamanho variável não definido (`[ ]`).
- Nas definições, parâmetros devem ser nomeados.

Parâmetros e argumentos — restrições

- Parâmetros não podem conter especificador de classe, exceto `register`.
- Parâmetros não podem ser iniciados.
- Nas declarações, além dos tipos completos, os parâmetros podem ser de um tipo incompleto, ou de um tipo vetor de tamanho variável não especificado (`[*]`).
- Nas definições, os parâmetros só podem ser de um tipo completo, ou de um tipo vetor de tamanho variável não definido (`[ ]`).
- Nas definições, parâmetros devem ser nomeados.

Observação:

As restrições referentes a ponteiros e vetores devem ser discutidas após o estudo desses tópicos!

Parâmetros e argumentos — restrições

Exemplo

Declaração/Definição	Erro
`int funA(float a = 3.4f, int);`	
`int funA(float a, int b = 4) { /* codigo omitido */ }`	
`int funA(float a, int) { /* codigo omitido */ }`	
`int funA(static int a) { /* codigo omitido */ }`	

```
int funA(float a = 3.4f, int);

int funA(float a, int b = 4) {
 /* codigo omitido */
}

int funA(float a, int) {
 /* codigo omitido */
}

int funA(static int a) {
 /* codigo omitido */
}
```

Parâmetros e argumentos — restrições

Exemplo

Declaração/Definição	Erro
`int funA(float a = 3.4f, int);`	parâmetros com iniciação.

```
int funA(float a, int b = 4) {
 /* codigo omitido */
}

int funA(float a, int) {
 /* codigo omitido */
}

int funA(static int a) {
 /* codigo omitido */
}
```

Parâmetros e argumentos — restrições

Exemplo

Declaração/Definição	Erro
`int funA(float a = 3.4f, int);`	parâmetros com iniciação.
`int funA(float a, int b = 4) { /* codigo omitido */ }`	parâmetros com iniciação.
`int funA(float a, int) { /* codigo omitido */ }`	
`int funA(static int a) { /* codigo omitido */ }`	

Parâmetros e argumentos — restrições

Exemplo

Declaração/Definição	Erro
`int funA(float a = 3.4f, int);`	parâmetros com iniciação.
`int funA(float a, int b = 4) { /* codigo omitido */ }`	parâmetros com iniciação.
`int funA(float a, int) { /* codigo omitido */ }`	parâmetro não-nomeado.
`int funA(static int a) { /* codigo omitido */ }`	

Parâmetros e argumentos — restrições

Exemplo

Declaração/Definição	Erro
`int funA(float a = 3.4f, int);`	parâmetros com iniciação.

`int funA(float a, int b = 4) {`
` /* codigo omitido */`
`}`
	parâmetros com iniciação.

`int funA(float a, int) {`
` /* codigo omitido */`
`}`
	parâmetro não-nomeado.

`int funA(static int a) {`
` /* codigo omitido */`
`}`
	classe diferente de `register`.

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	
`int funA(int []);`	
`int funA(int a[*]) { /* codigo omitido */ }`	
`int funA(int a[]) { /* codigo omitido */ }`	
`int funA(struct reg);`	
`int funA(struct reg a) { /* codigo omitido */ }`	

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	Declaração válida.
`int funA(int []);`	
`int funA(int a[*]) { /* codigo omitido */ }`	
`int funA(int a[]) { /* codigo omitido */ }`	
`int funA(struct reg);`	
`int funA(struct reg a) { /* codigo omitido */ }`	

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	Declaração válida.
`int funA(int []);`	Declaração válida.

`int funA(int a[*]) {`
` /* codigo omitido */`
`}`	

`int funA(int a[]) {`
` /* codigo omitido */`
`}`	

`int funA(struct reg);`	

`int funA(struct reg a) {`
` /* codigo omitido */`
`}`	

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	Declaração válida.
`int funA(int []);`	Declaração válida.

`int funA(int a[*]) {`
` /* codigo omitido */`
`}`
	Definição inválida (`[*]` apenas em declarações).

`int funA(int a[]) {`
` /* codigo omitido */`
`}`	

`int funA(struct reg);`	

`int funA(struct reg a) {`
` /* codigo omitido */`
`}`	

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	Declaração válida.
`int funA(int []);`	Declaração válida.

`int funA(int a[*]) {`
` /* codigo omitido */`
`}`
	Definição inválida ([*] apenas em declarações).

`int funA(int a[]) {`
` /* codigo omitido */`
`}`
	Definição válida.

`int funA(struct reg);`	

`int funA(struct reg a) {`
` /* codigo omitido */`
`}`	

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	Declaração válida.
`int funA(int []);`	Declaração válida.
`int funA(int a[*]) { /* codigo omitido */ }`	Definição inválida (`[*]` apenas em declarações).
`int funA(int a[]) { /* codigo omitido */ }`	Definição válida.
`int funA(struct reg);`	Declaração válida.
`int funA(struct reg a) { /* codigo omitido */ }`	

Parâmetros e argumentos — restrições

Declaração/Definição	Validade
`int funA(int [*]);`	Declaração válida.
`int funA(int []);`	Declaração válida.

`int funA(int a[*]) {`
` /* codigo omitido */`
`}`
	Definição inválida (`[*]` apenas em declarações).

`int funA(int a[]) {`
` /* codigo omitido */`
`}`
	Definição válida.

`int funA(struct reg);`	Declaração válida.

`int funA(struct reg a) {`
` /* codigo omitido */`
`}`
	Definição inválida (parâmetro com tipo incompleto).

Chamadas a funções

- Os parâmetros declarados como função retornando $\langle T \rangle$ e vetor de $\langle T \rangle$ têm seus tipos ajustados para ponteiro para função retornando $\langle T \rangle$ e ponteiro para $\langle T \rangle$, respectivamente.
- As expressões usadas como argumentos são avaliadas e seus valores atribuídos aos parâmetros correspondentes
- A avaliação dos argumentos não é sequenciada
- A execução inicia apenas após a avaliação e atribuição de todos os argumentos.
- O modo como os valores são atribuídos aos parâmetros depende da chamada ocorrer dentro ou fora do escopo de um protótipo da função.

Chamadas a funções

No escopo de um protótipo

- Os argumentos são convertidos implicitamente (como em uma atribuição) no tipo dos parâmetros correspondentes.

Chamadas a funções

No escopo de um protótipo

- Os argumentos são convertidos implicitamente (como em uma atribuição) no tipo dos parâmetros correspondentes.

Fora do escopo de um protótipo

- Os argumentos são promovidos segundo a seguinte *promoção padrão dos argumentos*:
  - A promoção inteira é aplicada a cada argumento do tipo inteiro.
  - Os argumentos do tipo `float` são promovidos para `double`.

Chamadas a funções

No escopo de um protótipo

- Os argumentos são convertidos implicitamente (como em uma atribuição) no tipo dos parâmetros correspondentes.

Fora do escopo de um protótipo

- Os argumentos são promovidos segundo a seguinte *promoção padrão dos argumentos*:
  - A promoção inteira é aplicada a cada argumento do tipo inteiro.
  - Os argumentos do tipo `float` são promovidos para `double`.

O comportamento é indefinido se

- A quantidade de argumentos é diferente da quantidade de parâmetros.
- O tipo de um argumento (após a promoção) não é compatível com o tipo do parâmetro correspondente.

Valor de retorno

O comando `return` finaliza a execução da função produzindo o valor de retorno que resulta da avaliação da sua expressão.

Exemplo

`return 2 * y;` Retorna com o valor $2 \times \texttt{y}$.

Valor de retorno

- O tipo do valor produzido pela expressão deve ser compatível com o tipo declarado.
- Se o tipo declarado é `void`, o comando `return` não deve possuir expressão de retorno.
- O tipo declarado para o valor de retorno não pode ser um tipo função ou vetor.
- Em uma definição de função o tipo declarado para o valor de retorno deve ser completo.

Funções recursivas

Uma função é recursiva quando chama ela mesma, direta ou indiretamente.

Recursividade direta, quando uma função chama ela mesma.

Recursividade indireta, quando uma função chama outra função que chama outra, em uma sequência que eventualmente resulta em uma chamada à função inicial.

Funções recursivas

São adequadas quando o problema pode ser expresso de modo recursivo

Função potência

Para $b > 0$:

$$a^b = \begin{cases} a & \text{Se } b = 1 \qquad \text{(condição de parada)} \\ a \times a^{(b-1)} & \end{cases}$$

Funções recursivas

São adequadas quando o problema pode ser expresso de modo recursivo

Função potência

Para $b > 0$:

$$a^b = \begin{cases} a & \text{Se } b = 1 \qquad \text{(condição de parada)} \\ a \times a^{(b-1)} & \end{cases}$$

Função fatorial

Para $a \geq 0$:

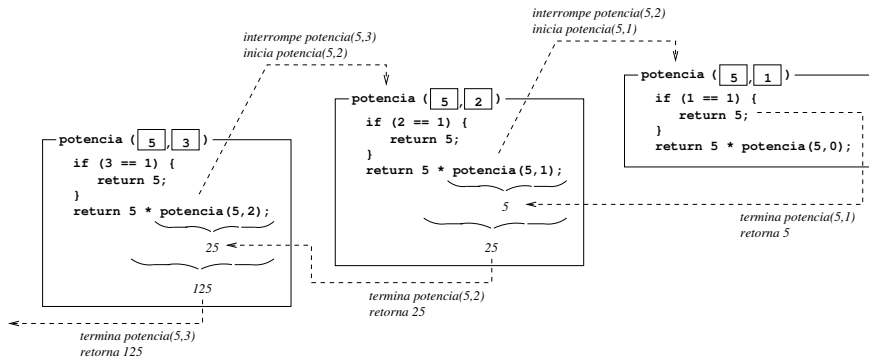
$$a! = \begin{cases} 1 & \text{Se } a = 0 \qquad \text{(condição de parada)} \\ a \times (a-1)! & \end{cases}$$

Funções recursivas

Exemplo

O programa ao lado calcula a função potência de modo recursivo.

```
#include <stdio.h>
int potencia(int, int);
int main(void) {
  printf("%d\n", potencia(5, 3));
  return 0;
}
int potencia(int a, int b) {
  if (b == 1) {
    return a;
  }
  return a * potencia(a, --b);
}
```

Funções recursivas

Número variável de parâmetros

O uso de reticências na declaração de funções especifica uma quantidade variável de parâmetros.

```
float funA(int a, ...)
```

Função retornando `float` com no mínimo um parâmetro. O primeiro parâmetro é do tipo `int` e os demais não são definidos.

```
void funB(char a, long b, ...)
```

Função sem valor de retorno com no mínimo dois parâmetros. O primeiro é do tipo `char`, o segundo é do tipo `long` e os demais não são definidos.

Acessando os argumentos adicionais

O cabeçalho `stdarg.h` possui macros para acessar os argumentos adicionais de uma função com argumentos variáveis.

Acessando os argumentos adicionais

O cabeçalho `stdarg.h` possui macros para acessar os argumentos adicionais de uma função com argumentos variáveis.

1. Define-se uma lista que receberá os argumentos correspondentes aos parâmetros não declarados:
   `va_list` ⟨*listaArg*⟩;

Acessando os argumentos adicionais

O cabeçalho `stdarg.h` possui macros para acessar os argumentos adicionais de uma função com argumentos variáveis.

1. Define-se uma lista que receberá os argumentos correspondentes aos parâmetros não declarados:
   `va_list` ⟨*listaArg*⟩`;`
2. Inicia-se a lista de argumentos usando a macro `va_start` e informando a identificação do último parâmetro declarado:
   `va_start(`⟨*listaArg*⟩`,` ⟨*IdentUltimoPar*⟩`);`

Acessando os argumentos adicionais

O cabeçalho `stdarg.h` possui macros para acessar os argumentos adicionais de uma função com argumentos variáveis.

1. Define-se uma lista que receberá os argumentos correspondentes aos parâmetros não declarados:
   `va_list` ⟨*listaArg*⟩`;`
2. Inicia-se a lista de argumentos usando a macro `va_start` e informando a identificação do último parâmetro declarado:
   `va_start(`⟨*listaArg*⟩`,` ⟨*IdentUltimoPar*⟩`);`
3. Usa-se a macro `va_arg` para obter o próximo argumento da lista como um valor do tipo ⟨*tipoArg*⟩:
   `va_arg(`⟨*listaArg*⟩`,` ⟨*tipoArg*⟩`);`

Acessando os argumentos adicionais

O cabeçalho `stdarg.h` possui macros para acessar os argumentos adicionais de uma função com argumentos variáveis.

1. Define-se uma lista que receberá os argumentos correspondentes aos parâmetros não declarados:
   `va_list` ⟨*listaArg*⟩`;`
2. Inicia-se a lista de argumentos usando a macro `va_start` e informando a identificação do último parâmetro declarado:
   `va_start(`⟨*listaArg*⟩`,` ⟨*IdentUltimoPar*⟩`);`
3. Usa-se a macro `va_arg` para obter o próximo argumento da lista como um valor do tipo ⟨*tipoArg*⟩:
   `va_arg(`⟨*listaArg*⟩`,` ⟨*tipoArg*⟩`);`
4. Ao final, libera-se a lista de argumentos com a macro `va_end`:
   `va_end(`⟨*listaArg*⟩`);`

Número variável de parâmetros

Exemplo

A função funA ao lado recebe um número variável de argumentos:

- O primeiro indica quantos virão a seguir.
- Todos os demais, exceto o último são do tipo int.
- O último é do tipo double

```
void funA(int qtd, ...) {
  va_list lpar;
  va_start(lpar, qtd);
  int argc;
  double argd ;
  for (int i = 0; i < qtd - 1; i++) {
    argc = va_arg(lpar, int);
    printf("%c ", argc);
  }
  if (qtd > 0) {
    argd = va_arg(lpar, double);
    printf("%f\n", argd);
  }
  va_end(lpar);
}
```

Desvios não locais

A macro `setjmp` e a função `longjmp` (declaradas em `setjmp.h`) são usadas em conjunto para implementar desvios não locais:

- `setjmp` salva o ambiente de execução.
- `longjmp` retorna ao ponto onde o ambiente de execução foi salvo.

Desvios não locais

```
int setjmp(jmp_buf amb)
```

Salva o ambiente de execução na área de armazenamento temporário `amb`. A macro pode ser executada a partir do fluxo normal de execução ou em decorrência de uma chamada a `longjmp`.

Valor de retorno. Zero, se executada a partir do fluxo normal de execução. O valor de retorno quando a macro é executada em decorrência de uma chamada a `longjmp` é igual ao argumento fornecido à função `longjmp` ou 1, se esse argumento for igual a 0.

Desvios não locais

```
_Noreturn void longjmp(jmp_buf amb, int res)
```

Restaura o ambiente de execução armazenado em `amb`, causa o desvio para o ponto de chamada da função `setjmp` que salvou o ambiente `amb` e define `res` como o valor resultante dessa nova chamada a `setjmp`.

Valor de retorno. Não tem.

Desvios não locais

Exemplo. Qual o comportamento do programa se o valor lido for -1?

```
#include <stdio.h>
#include <setjmp.h>
void funA(int);
void funB(int);
jmp_buf estado;
int main(void) {
  int i = 0;
  printf("inicio prog\n");
  (void)setjmp(estado);
  printf("Valor de i: ");
  scanf("%d", &i);
  if (i < 2) {
    funA(i);
  }
  printf("fim prog\n");
  return 0;
}
void funA(int x) {
  printf("inicio funA\n");
  if (x > 0) {
    longjmp(estado, 2);
  }
  funB(2 * x);
  printf("fim funA\n");
}
void funB(int y) {
  printf("inicio funB\n");
  if (y < 0) {
    longjmp(estado, 4);
  }
  printf("fim funB\n");
}
```

Desvios não locais

Exemplo. Qual o comportamento do programa se o valor lido for -1?

O programa volta à função `main`, para uma nova leitura, diretamente da função `funB`, sem executar os retornos convencionais.

```
inicio prog
Valor de i:  -1
inicio funA
inicio funB
Valor de i:
```

A função main

`int main(void)`

Inicia a execução do programa.

Valor de retorno. Valor inteiro indicando o estado do término da execução.

`int main(int qtd_arg, char *args[])`

Inicia a execução do programa armazenando em `qtd_arg` a quantidade de argumentos da linha de comando e em `args` os argumentos fornecidos.

Valor de retorno. Valor inteiro indicando o estado do término da execução.

Classe de armazenamento

static. Modo de alocação estático. Ligação interna.

extern. Modo de alocação estático. Ligação externa, exceto se houver no mesmo escopo uma declaração prévia com ligação interna, caso em que a ligação será interna.

Sem qualificador. Modo de alocação e ligação determinados como se tivesse sido declarada com o qualificador `extern`.

Classe de armazenamento

Exemplo

Unid. compilação 1

```
#include <stdio.h>
static void funA(void);
void funB(void);
extern void funC(void);
void funD(void);
int main(void) {
  funA(); funB(); funC(); funD();
  return 0;
}
static void funA(void) {
  printf("funA(1)\n");
}
void funD(void) {
  printf("funD\n");
}
```

Unid. compilação 2

```
#include <stdio.h>
static void funA(void);
void funB(void);
void funC(void);
extern void funD(void);
void funB(void) {
  printf("funB\n");
  funA();
}
extern void funA(void) {
  printf("funA(2)\n");
}
extern void funC(void) {
  printf("funC\n");
  funD();
}
```

Funções em linha

- O especificador `inline` orienta o compilador a inserir o código da função no local da sua chamada.
- Qualquer função com ligação interna pode ser declarada em linha.
- Uma função com ligação externa declarada em linha deve ser definida na mesma unidade de compilação que a declaração em linha.

Funções em linha

Exemplo

```
#include <stdio.h>
inline static int dobro(int);
int main(void) {
  printf("%d\n", dobro(23));
  return 0;
}
static int dobro(int a) {
  return 2 * a;
}
```

```
#include <stdio.h>
static int dobro(int);
int main(void) {
  printf("%d\n", dobro(23));
  return 0;
}
inline static int dobro(int a) {
  return 2 * a;
}
```

Funções em linha

Exemplo

```
#include <stdio.h>
inline static int dobro(int);
int main(void) {
  printf("%d\n", dobro(23));
  return 0;
}
static int dobro(int a) {
  return 2 * a;
}
```

```
#include <stdio.h>
static int dobro(int);
int main(void) {
  printf("%d\n", dobro(23));
  return 0;
}
inline static int dobro(int a) {
  return 2 * a;
}
```

- A chamada a `dobro` no programa à direita pode não ser colocada em linha porque não está no escopo de uma declaração em linha.
- Muitas outras circunstâncias podem fazer com que uma função não seja colocada em linha.

O tipo de uma função

O tipo de uma função é caracterizado pelo tipo do seu valor de retorno e dos seus parâmetros.

Declaração	Tipo
`int funA()`	`int ()`
`int funB(void)`	`int (void)`
`char *funC(int, float)`	`char *(int, float)`
`struct reg funD(int)`	`struct reg (int)`

O tipo de uma função

O tipo de uma função é caracterizado pelo tipo do seu valor de retorno e dos seus parâmetros.

Declaração	Tipo
`int funA()`	`int ()`
`int funB(void)`	`int (void)`
`char *funC(int, float)`	`char *(int, float)`
`struct reg funD(int)`	`struct reg (int)`

Função retornando `int`.

O tipo de uma função

O tipo de uma função é caracterizado pelo tipo do seu valor de retorno e dos seus parâmetros.

Declaração	Tipo
`int funA()`	`int ()`
`int funB(void)`	`int (void)`
`char *funC(int, float)`	`char *(int, float)`
`struct reg funD(int)`	`struct reg (int)`

Função (`void`) retornando `int`.

O tipo de uma função

O tipo de uma função é caracterizado pelo tipo do seu valor de retorno e dos seus parâmetros.

Declaração	Tipo
`int funA()`	`int ()`
`int funB(void)`	`int (void)`
`char *funC(int, float)`	`char *(int, float)`
`struct reg funD(int)`	`struct reg (int)`

Função de `int` e `float` retornando ponteiro para `char`.

O tipo de uma função

O tipo de uma função é caracterizado pelo tipo do seu valor de retorno e dos seus parâmetros.

Declaração	Tipo
`int funA()`	`int ()`
`int funB(void)`	`int (void)`
`char *funC(int, float)`	`char *(int, float)`
`struct reg funD(int)`	`struct reg (int)`

Função de `int` retornando `struct reg`.

Definição de tipo função

Um tipo função pode ser definido com o operador `typedef` usando-se a declaração do novo tipo como se fosse a declaração de um protótipo de função.

Exemplo

```
typedef int funA_t(int)
typedef void funB_t(float *, int)
```

O tipo `funA_t` é sinônimo de `int (int)`.
O tipo `funB_t` é sinônimo de `void (float *, int)`.

Definição de tipo função

Um tipo função pode ser definido com o operador `typedef` usando-se a declaração do novo tipo como se fosse a declaração de um protótipo de função.

Exemplo

```
typedef int funA_t(int)
typedef void funB_t(float *, int)
```

O tipo `funA_t` é sinônimo de `int (int)`.
O tipo `funB_t` é sinônimo de `void (float *, int)`.

Os novos tipos podem ser usados em declarações:

`funA_t fun;` declara `fun` como do tipo `funA_t`.

`funB_t fun;` declara `fun` como do tipo `funB_t`.

Compatibilidade de tipos

Dois tipos função são compatíveis

- se possuem tipos de valor de retorno compatíveis
- e, se ambos possuem lista de parâmetros, as seguintes condições são satisfeitas:
  - a quantidade de parâmetros deve ser igual,
  - os tipos dos parâmetros correspondentes devem ser compatíveis, e
  - se uma lista de parâmetros contém reticências, a outra também deve conter.

Ponteiro para função

Se `f_ptr` é um ponteiro para função do tipo $\langle T \rangle$ `(void)`, então

- `f_ptr()` e `(*f_ptr)()` causam a execução da função apontada pelo ponteiro;
- o valor resultante é do tipo $\langle T \rangle$.

Ponteiro para função

Exemplo

O programa ao lado executa a função

- funA, se o valor lido for igual a 1 ou
- funB, se o valor lido for igual a 2.

```
#include <stdio.h>
void funA(char);
void funB(char);
typedef void fun_t(char);
int main(void) {
  int op;
  fun_t *f[2] = {funA, funB};
  printf("Operacao (1 ou 2): ");
  scanf("%d", &op);
  if ((op == 1) || (op == 2)) {
    f[op - 1]('x');
  }
  return 0;
}
void funA(char c) {
  printf("funA: %c\n", c);
}
void funB(char c) {
  printf("funB: %c\n", c);
}
```

Controlando o término da execução

- Término normal
  - O fluxo da execução atinge o fim da função `main`.
  - A função `main` é finalizada pela execução do comando `return`.
  - As funções `exit`, `quick_exit` ou `_Exit` são executadas.
- Término anormal
  - A execução é interrompida pela ocorrência de um erro de execução não recuperável, que pode ser lançado pelo ambiente ou pelas funções `raise` ou `abort`.

Controlando o término da execução

As seguintes ações podem ser executadas por um programa por ocasião de seu término, dependendo do modo como ele é finalizado:

1. Executar as funções de término registradas pelas funções `atexit` ou `at_quick_exit`.
2. Gravar nos arquivos de saída os dados ainda não gravados que estejam em suas áreas de armazenamento temporário.
3. Fechar os arquivos ainda abertos.
4. Remover os arquivos temporários.

As macros `EXIT_SUCCESS` e `EXIT_FAILURE`, bem como as funções `atexit`, `at_quick_exit`, `exit`, `quick_exit`, `_Exit` e `abort` são declaradas no cabeçalho `stdlib.h`.

Controlando o término da execução

```
int atexit(void (*fun)(void))
```

Registra a função apontada por `fun`, que será executada se o programa terminar normalmente. A função `fun` deve ser definida como uma função sem parâmetros retornando `void`.

Valor de retorno. Zero, se o registro é bem sucedido, ou um valor diferente de 0, em caso de falha.

Controlando o término da execução

```
_Noreturn void exit(int estado)
```

Causa o término normal do programa lançando o código `estado` para ser capturado pelo ambiente, e.g. por um roteiro de execução a partir do qual o programa foi iniciado.

Valor de retorno. Não tem.

Controlando o término da execução

```
_Noreturn void _Exit(int estado)
```

Causa o término normal do programa lançando o código `estado` para ser capturado pelo ambiente de execução. Entretanto, nenhuma função registrada com `atexit` ou `signal` é chamada. A gravação dos dados que estejam nas áreas de armazenamento temporário associadas às operações de saída, o fechamento dos arquivos abertos e a remoção dos arquivos temporários é dependente da implementação.

Valor de retorno. Não tem.

Controlando o término da execução

```
_Noreturn void abort(void)
```

Lança um sinal SIGABRT, que causará o término anormal do programa se não for capturado e tratado.

Valor de retorno. Não tem.

Controlando o término da execução

```
_Noreturn void abort(void)
```

Lança um sinal `SIGABRT`, que causará o término anormal do programa se não for capturado e tratado.

Valor de retorno. Não tem.

Observação:

Em um término anormal não há garantia de que as áreas de armazenamento temporário sejam esvaziadas, os arquivos abertos sejam fechados e os temporários removidos.

Executando comandos do sistema

```
int system(const char *comando)
```

Envia a cadeia apontada por `comando` para execução pelo processador de comandos, ou determina se o ambiente de execução possui um processador de comandos, se a cadeia `comando` é nula. O comportamento é indefinido se a cadeia `comando` for enviada em um ambiente que não possua processador de comandos.

Valor de retorno. Se a cadeia `comando` é nula, retorna um valor diferente de zero, se o ambiente possui um processador de comandos, ou zero, em caso contrário. Se `comando` é diferente de nulo, o valor retornado depende da implementação.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

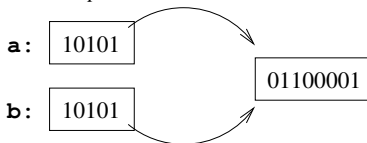
Ponteiros e vetores

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Ponteiros

- As variáveis do tipo ponteiro para $\langle T \rangle$ armazenam endereços de memória.

- Os valores armazenados nesses endereços são interpretados como valores do tipo $\langle T \rangle$ (quando acessados por meio da variável ponteiro para $\langle T \rangle$ que designa o endereço).

Declaração de ponteiros

$$\langle DeclPonteiro \rangle ::= \mathbf{*}\ [\ \langle QualifTipo \rangle\ ]$$
$$\qquad\qquad\qquad |\ \ \mathbf{*}\ [\ \langle QualifTipo \rangle\ ]\ \langle DeclPonteiro \rangle$$

Declarações válidas:

`int *ptr_a;`	Declara `ptr_a` do tipo ponteiro para `int`.
`int *ptr_a, ptr_b;`	Declara as variáveis `ptr_a`, do tipo ponteiro para `int`, e `ptr_b`, do tipo `int`.

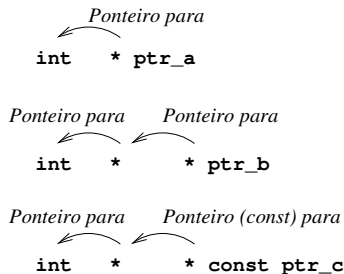
Declaração de ponteiros

Declarações válidas:

`int **ptr_a;`	Declara `ptr_a` do tipo ponteiro para ponteiro para `int`.
`int * const ptr_a;`	Declara `ptr_a` do tipo ponteiro (constante) para `int`. O conteúdo de `ptr_a` não pode ser modificado, o conteúdo apontado por `ptr_a` pode.
`const int * ptr_a;`	Declara `ptr_a` do tipo ponteiro para `const int`. O conteúdo de `ptr_a` pode ser modificado, o conteúdo apontado por `ptr_a` não pode.

Declaração de ponteiros

Exemplo

As declarações
`int *ptr_a;`
`int **ptr_b;` e
`int ** const ptr_c;`
são interpretadas do seguinte modo:

Declaração de ponteiros

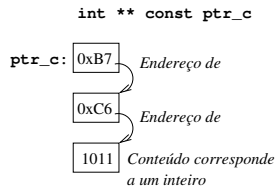
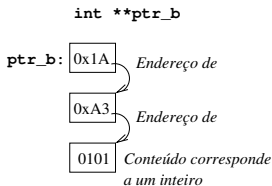
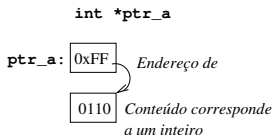
Exemplo

As declarações

`int *ptr_a;`

`int **ptr_b;` e

`int ** const ptr_c;`

são interpretadas do seguinte modo:

Operador de endereço

O operador &, aplicado a uma expressão que designa uma localização de memória, resulta no endereço da localização designada pela expressão.

Operador de endereço

O operador `&`, aplicado a uma expressão que designa uma localização de memória, resulta no endereço da localização designada pela expressão.

Exemplo

- `&aluno` resulta no endereço da variável `aluno`
- `&(notas[i])` resulta no endereço designado pela expressão `(notas[i])`.

Operador de endereço

Exemplo

O programa ao lado imprime os endereços armazenados nas variáveis `ptr_a` e `ptr_b`.

```
#include <stdio.h>
int main(void) {
  int x = 97;
  int *ptr_a;
  char *ptr_b;
  ptr_a = &x;
  ptr_b = &x;
  printf("%p %p\n", (void *)ptr_a,
                    (void *)ptr_b);
  return 0;
}
```

Operador de acesso indireto

O operador *, quando aplicado a uma variável do tipo ponteiro, resulta na referência ao endereço apontado pelo ponteiro, que pode ser usada:

- para obter o conteúdo referido ou
- como o operando esquerdo do operador de atribuição

Operador de acesso indireto

O operador *, quando aplicado a uma variável do tipo ponteiro, resulta na referência ao endereço apontado pelo ponteiro, que pode ser usada:

- para obter o conteúdo referido ou
- como o operando esquerdo do operador de atribuição

Exemplo

Se `ptr_a` é uma variável do tipo ponteiro para `int`, então:

`ptr_a`	resulta no valor armazenado em `ptr_a`: um endereço.

Operador de acesso indireto

O operador *, quando aplicado a uma variável do tipo ponteiro, resulta na referência ao endereço apontado pelo ponteiro, que pode ser usada:

- para obter o conteúdo referido ou
- como o operando esquerdo do operador de atribuição

Exemplo

Se `ptr_a` é uma variável do tipo ponteiro para `int`, então:

`ptr_a`	resulta no valor armazenado em `ptr_a`: um endereço.
`*ptr_a`	resulta na referência ao espaço de memória apontado por `ptr_a`.

Operador de acesso indireto

O operador *, quando aplicado a uma variável do tipo ponteiro, resulta na referência ao endereço apontado pelo ponteiro, que pode ser usada:

- para obter o conteúdo referido ou
- como o operando esquerdo do operador de atribuição

Exemplo

Se `ptr_a` é uma variável do tipo ponteiro para `int`, então:

`ptr_a`	resulta no valor armazenado em `ptr_a`: um endereço.
`*ptr_a`	resulta na referência ao espaço de memória apontado por `ptr_a`.
`*ptr_a = 23;`	atribui o valor 23 ao espaço de memória apontado por `ptr_a`.

Operador de acesso indireto

O operador `*`, quando aplicado a uma variável do tipo ponteiro, resulta na referência ao endereço apontado pelo ponteiro, que pode ser usada:

- para obter o conteúdo referido ou
- como o operando esquerdo do operador de atribuição

Exemplo

Se `ptr_a` é uma variável do tipo ponteiro para `int`, então:

`ptr_a`	resulta no valor armazenado em `ptr_a`: um endereço.
`*ptr_a`	resulta na referência ao espaço de memória apontado por `ptr_a`.
`*ptr_a = 23;`	atribui o valor 23 ao espaço de memória apontado por `ptr_a`.
`printf("%d", *ptr_a);`	imprime o conteúdo do espaço de memória apontado por `ptr_a`.

Operador de acesso indireto

Exemplo

O que é impresso pelo código ao lado?

```
#include <stdio.h>
int main(void) {
  int x = 97;
  int y = 76;
  int *ptr_x = &x;
  int *ptr_y = &y;
  printf("%d %d\n", *ptr_x, *ptr_y);
  *ptr_x = 123 + *ptr_y;
  (*ptr_y)++;
  printf("%d %d\n", x, y);
  return 0;
}
```

Operador de acesso indireto

Exemplo

O que é impresso pelo código ao lado?

```
#include <stdio.h>
int main(void) {
  int x = 97;
  int y = 76;
  int *ptr_x = &x;
  int *ptr_y = &y;
  printf("%d %d\n", *ptr_x, *ptr_y);
  *ptr_x = 123 + *ptr_y;
  (*ptr_y)++;
  printf("%d %d\n", x, y);
  return 0;
}
```

Resposta:

97 76

199 77

Relação entre * e &

As seguintes equivalências são válidas:

- `&*ptr` ≡ `ptr`
- `*&var_a` ≡ `var_a`
- `&(vet[x])` ≡ `((vet) + (x))`

Ponteiros para funções

Na declaração de ponteiros para funções deve ser observado que o operador () tem maior precedência que o declarador de ponteiro *:

Ponteiros para funções

Na declaração de ponteiros para funções deve ser observado que o operador `()` tem maior precedência que o declarador de ponteiro `*`:

`void *fun(void);`	Declara `fun` como uma função sem argumentos retornando um ponteiro para `void`.
`void (*fun)(void);`	Declara `fun` como um ponteiro para uma função sem argumentos retornando `void`.

Ponteiros para funções

Na declaração de ponteiros para funções deve ser observado que o operador `()` tem maior precedência que o declarador de ponteiro `*`:

`void *fun(void);`	Declara `fun` como uma função sem argumentos retornando um ponteiro para `void`.
`void (*fun)(void);`	Declara `fun` como um ponteiro para uma função sem argumentos retornando `void`.
`int *fun(int);`	Declara `fun` como uma função de `int` retornando um ponteiro para `int`.
`int (*fun)(int);`	Declara `fun` como um ponteiro para uma função de `int` retornando `int`.

Vetores

Sequências de elementos de um dado tipo, armazenados em posições contíguas da memória, distribuídos em um número predeterminado de dimensões.

Declaração de vetores

⟨*DeclaradorVetor*⟩ ::= ⟨*Identificador*⟩ ⟨*DeclDim*⟩ { ⟨*DeclDim*⟩ }

⟨*DeclDim*⟩ ::= **[** [⟨*ListaQualifTipo*⟩] [⟨*QtdElmDim*⟩] **]**
| **[** **static** [⟨*ListaQualifTipo*⟩] ⟨*QtdElmDim*⟩ **]**
| **[** ⟨*ListaQualifTipo*⟩ **static** ⟨*QtdElmDim*⟩ **]**
| **[** [⟨*ListaQualifTipo*⟩] ***** **]**

⟨*ListaQualifTipo*⟩ ::= ⟨*QualifTipo*⟩ | ⟨*ListaQualifTipo*⟩ ⟨*QualifTipo*⟩

⟨*QtdElmDim*⟩ ::= Expressão do tipo inteiro definindo o tamanho da dimensão do vetor.

Declaração de vetores

`int alunos[3];`	Vetor de `int`, de uma dimensão, com 3 elementos.
`double alunos[2][3];`	Vetor bidimensional de `double`, com 2 elementos na primeira dimensão e 3 na segunda. De fato, vetor de uma dimensão com 2 elementos do tipo vetor de uma dimensão.

Tipo dos vetores

- Quando os elementos de um vetor são de um tipo $\langle T \rangle$, diz-se que a variável que o declara é um *vetor de* $\langle T \rangle$.
- O tipo do vetor é $\langle T \rangle$[]...[] (incluindo a especificação de cada dimensão).
- Os vetores de $\langle T \rangle$ podem ter seu tipo:

  Completo. Quando a quantidade de elementos é definida.

  Incompleto. Quando a quantidade de elementos não é definida (a expressão da quantidade é omitida).

  Variável. Quando a quantidade de elementos não é constante.

  Variável (com quantidade não especificada). A quantidade é caracterizada por um asterisco.

Tipo dos vetores

Exemplo

Qual o tipo dos seguintes vetores?

```
int alunos[3];

double alunos[2][3];

char *alunos[];

char *alunos[2 + x];

long [*]
```

Tipo dos vetores

Exemplo

Qual o tipo dos seguintes vetores?

`int alunos[3];` Vetor do tipo `int[3]`. Tipo completo.

`double alunos[2][3];`

`char *alunos[];`

`char *alunos[2 + x];`

`long [*]`

Tipo dos vetores

Exemplo

Qual o tipo dos seguintes vetores?

`int alunos[3];` Vetor do tipo `int[3]`. Tipo completo.

`double alunos[2][3];` Vetor do tipo `double[2][3]`. Tipo completo.

`char *alunos[];`

`char *alunos[2 + x];`

`long [*]`

Tipo dos vetores

Exemplo

Qual o tipo dos seguintes vetores?

`int alunos[3];` — Vetor do tipo `int[3]`. Tipo completo.

`double alunos[2][3];` — Vetor do tipo `double[2][3]`. Tipo completo.

`char *alunos[];` — Vetor do tipo `char *[]`. Tipo incompleto.

`char *alunos[2 + x];`

`long [*]`

Tipo dos vetores

Exemplo

Qual o tipo dos seguintes vetores?

`int alunos[3];`	Vetor do tipo `int[3]`. Tipo completo.
`double alunos[2][3];`	Vetor do tipo `double[2][3]`. Tipo completo.
`char *alunos[];`	Vetor do tipo `char *[]`. Tipo incompleto.
`char *alunos[2 + x];`	Vetor do tipo `char *[2 + x]`. Tipo variável (porém completo).
`long [*]`	

Tipo dos vetores

Exemplo

Qual o tipo dos seguintes vetores?

`int alunos[3];`	Vetor do tipo `int[3]`. Tipo completo.
`double alunos[2][3];`	Vetor do tipo `double[2][3]`. Tipo completo.
`char *alunos[];`	Vetor do tipo `char *[]`. Tipo incompleto.
`char *alunos[2 + x];`	Vetor do tipo `char *[2 + x]`. Tipo variável (porém completo).
`long [*]`	Vetor do tipo `long [*]`. Tipo variável (de tamanho não especificado, porém completo).

Atribuição e referência

- Os elementos de um vetor são modificados apenas individualmente, em atribuições da forma `vetA[indice] = 23`.
- A referência a um elemento é feita indicando-se o índice do elemento, entre os colchetes, após o nome do vetor:
  - o primeiro elemento possui índice 0,
  - o segundo, índice 1,
  - o vigésimo, índice 19, etc.
- Os elementos de um vetor, desde que completamente determinados, podem participar de qualquer operação compatível com o seu tipo.

Atribuição e referência

Exemplo

Considerando a declaração `int vet[150];` e considerando que a variável `x` tenha o valor 17, então:

`vet[3] = 23;`	Atribui o valor 23 ao quarto elemento de `vet`.
`vet[2 * x + 1] = 3.56;`	Atribui o valor 3 (conversão do valor $3,56$ do tipo `double` em um valor do tipo `int`) ao trigésimo sexto elemento de `vet`.
`vet[3]++;`	Adiciona 1 ao valor de `vet[3]`, isto é, ao quarto elemento de `vet`.
`--vet[3];`	Subtrai 1 do valor de `vet[3]`.
`(vet[3] > 2)`	Compara o valor de `vet[3]` com o inteiro 2.

Vetores unidimensionais

Exemplo

O programa ao lado lê 10 números inteiros, armazenando-os em um vetor de `int`, e imprime os números lidos, após a leitura.

```
#include <stdio.h>
int main(void) {
  int vetnum[10];
  int num;
  for (int i = 0; i < 10; i++) {
    printf("digite elm %d: ", i);
    scanf("%d", &num);
    vetnum[i] = num;
  }
  for (int i = 0; i < 10; i++) {
    printf("%d ", vetnum[i]);
  }
  return 0;
}
```

Vetores unidimensionais — tamanho variável

Exemplo

O programa ao lado lê uma quantidade de números inteiros especificada pelo usuário, armazenando-os em um vetor de `int`, e imprime os números lidos, após a leitura.

```
#include <stdio.h>
int main(void) {
  int qtd;
  printf("Digite a qtd elms: ");
  scanf("%d", &qtd);
  int vetnum[qtd];
  for (int i = 0; i < qtd; i++) {
    printf("digite elm %d: ", i);
    scanf("%d", &vetnum[i]);
  }
  for (int i = 0; i < qtd; i++) {
    printf("%d ", vetnum[i]);
  }
  return 0;
}
```

Vetores unidimensionais — tamanho variável

Exemplo

O programa ao lado lê uma quantidade de números inteiros especificada pelo usuário, armazenando-os em um vetor de `int`.
Após a leitura a função `imp_vet` é chamada para imprimir o vetor lido.

```
#include <stdio.h>
void imp_vet(int, int [*]);
int main(void) {
  int qtd;
  printf("Digite a qtd elms: ");
  scanf("%d", &qtd);
  int vetnum[qtd];
  for (int i = 0; i < qtd; i++) {
    printf("digite elm %d: ", i);
    scanf("%d", &vetnum[i]);
  }
  imp_vet(qtd, vetnum);
  return 0;
}
void imp_vet(int x, int vet[x]) {
  for (int i = 0; i < x; i++) {
    printf("%d ", vet[i]);
  }
}
```

Vetores unidimensionais — incompletos

Exemplo

prg_vetA.c

```
#include <stdio.h>
extern char estado[];
extern int  qtd;
void inicia_vet(char);
void imp_vet(char[]);
int main(void) {
  inicia_vet('a');
  imp_vet(estado);
  return 0;
}
void imp_vet(char v[]) {
  for (int i = 0; i < qtd; i++) {
    printf("%c ", v[i]);
  }
}
```

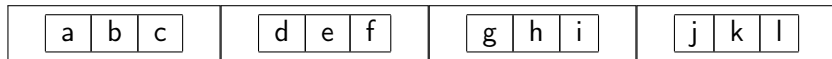
prg_vetB.c

```
#define TAM (10)
int  qtd = TAM;
char estado[TAM];
void inicia_vet(char c) {
  for (int i = 0; i < qtd; i++) {
    estado[i] = c++;
  }
}
```

Vetores multidimensionais

Os vetores multidimensionais são implementados em C como vetores de vetores, isto é, vetores cujos elementos são vetores.

O vetor bidimensional que corresponde à matriz

é implementado da seguinte forma:

Vetores multidimensionais

A referência aos elementos dos vetores multidimensionais pode ser

parcial, com a obtenção de um vetor componente, ou

total, com a obtenção de um elemento.

Vetores multidimensionais

A referência aos elementos dos vetores multidimensionais pode ser

parcial, com a obtenção de um vetor componente, ou

total, com a obtenção de um elemento.

Exemplo

Para uma matriz (vetor bidimensional) `A`:

As seguintes referências são válidas:

`A[0]` = | a | b | c | `A[2]` = | g | h | i | `A[3]` = | j | k | l |

`A[0][2]` = ‘c’   `A[2][1]` = ‘h’   `A[3][0]` = ‘j’

Vetores multidimensionais — declaração

`int alunos[3][2];`	Vetor de `int[2]` (vetor bidimensional de `int`). Tipo completo.
`char *alunos[][2][4];`	Vetor de `char *[2][4]` (vetor tridimensional de `char *`). Tipo incompleto.

Vetores multidimensionais — declaração

`long [][x];`	Vetor de `long[x]` (vetor bidimensional de `long`). Tipo incompleto.
`long [][*]`	Vetor de `long[*]` (vetor bidimensional de `long`). Tipo incompleto.

Vetores multidimensionais — declaração

`char *alunos[y][2 + x];` Vetor de `char *[2 + x]` (vetor bidimensional de `char *`). Tipo variável (porém, completo).

`short int alunos[3][2 + x];` Vetor de `short int[2 + x]` (vetor bidimensional de `short int`). Tipo variável (porém, completo).

Vetores multidimensionais — tipo

```
long *vetA[2][3]
```

Expressão	Tipo
`vetA`	`long *[2][3]`
`vetA[1]`	`long *[3]`
`vetA[1][2]`	`long *`

Vetores multidimensionais — tipo

`long *vetA[2][3]`

Expressão	Tipo
`vetA`	`long *[2][3]`
`vetA[1]`	`long *[3]`
`vetA[1][2]`	`long *`

`const char vetB[2][3][x]`

Expressão	Tipo
`vetB`	`const char[2][3][x]`
`vetB[1]`	`const char[3][x]`
`vetB[1][2]`	`const char[x]`
`vetB[1][0][1]`	`const char`

Vetores multidimensionais

Exemplo

O programa ao lado lê dois números, L e C, ambos maiores que 0, e depois lê, coluna a coluna, os números correspondentes aos elementos de uma matriz de ordem $L \times C$. Após a leitura, a matriz lida é impressa.

```
#include <stdio.h>
int main(void) {
  int L, C;
  do {
    printf("Digite a qtd de linhas: ");
    scanf("%d", &L);
  } while (L <= 0);
  do {
    printf("Digite a qtd de colunas: ");
    scanf("%d", &C);
  } while (C <= 0);
  int mat[L][C];
  printf("Digite, coluna a coluna, ");
  printf("os elementos de uma ");
  printf("matriz %d x %d\n", L, C);
```

Continua...

Vetores multidimensionais

Exemplo

Continuação.

```
  for (int j = 0; j < C; j++) {
    for (int i = 0; i < L; i++) {
      printf("elm (%d,%d):", (i+1), (j+1));
      scanf("%d", &mat[i][j]);
    }
  }
  for (int i = 0; i < L; i++) {
    for (int j = 0; j < C; j++) {
      printf("%d ", mat[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Vetores multidimensionais

Exemplo

A função ao lado recebe dois inteiros L e C e uma matriz de L linhas e C colunas, e imprime a matriz recebida.

```
void imp_vet(int L, int C, int m[L][C]) {
  for (int i = 0; i < L; i++) {
    for (int j = 0; j < C; j++) {
      printf("%d ", m[i][j]);
    }
    printf("\n");
  }
}
```

Iniciação de vetores

Os elementos do vetor são associados aos valores da lista de iniciação, segundo o seguinte processo:

1. Se o valor corrente pode ser atribuído ao elemento corrente, a atribuição é realizada e o processo prossegue com o próximo elemento do vetor e o próximo valor da lista, que serão os novos elemento e valor correntes.
2. Se o valor corrente é também uma lista, a atribuição de valor ao elemento corrente e a seus subelementos (se ele for um vetor) fica restrita aos valores da lista que corresponde ao valor corrente.
3. Se o elemento corrente é também um vetor, o processo prossegue recursivamente, associando o primeiro elemento desse vetor ao valor corrente, até que a atribuição seja realizada.

- Os elementos não iniciados de um vetor assumem o valor padrão que corresponde ao seu tipo.
- Os valores excedentes da lista de iniciação são ignorados.

Iniciação de vetores

Exemplo

```
char *fatorRH[8][2] =   {{"A", "+"}, {"A", "-"}, {"B", "+"}, {"B", "-"},
                          {"O", "+"}, {"AB", "-"}, {"O", "-"}};
```

1. fatorRH[0] e {"A", "+"}
   - fatorRH[0][0] $\leftarrow$ "A"
   - fatorRH[0][1] $\leftarrow$ "+"
2. fatorRH[1] e {"A", "-"}
   - fatorRH[1][0] $\leftarrow$ "A"
   - fatorRH[1][1] $\leftarrow$ "-"
3. fatorRH[2] e {"B", "+"}
   - fatorRH[2][0] $\leftarrow$ "B"
   - fatorRH[2][1] $\leftarrow$ "+"
4. fatorRH[3] e {"B", "-"}
5. ...

Iniciação de vetores

Exemplo

```
char *fatorRH[8][2] =  {"A", "+", "A", {"B", "+"}, {"B"}, {"O", "+"},
                        "O", "+", "AB", "-"};
```

1. fatorRH[0] e "A"
   - fatorRH[0][0] ← "A"
   - fatorRH[0][1] ← "+"
2. fatorRH[1] e "A"
   - fatorRH[1][0] ← "A"
   - fatorRH[1][1] ← {"B", "+"} ; fatorRH[1][1] ← "B"
3. fatorRH[2] e {"B"}
   - fatorRH[2][0] ← "B"
   - fatorRH[2][1] ← NULL
4. fatorRH[3] e {"O", "+"}
5. ...

Iniciação seletiva

Exemplo

```
char *fatorRH[8][2] =   {{"A", "+"}, [5]={"O", "-"}, {"AB", "+"},
                          [1]={"A", "-"}, {[1]= "+"}};
```

1. fatorRH[0] e {"A", "+"}
   - fatorRH[0][0] ← "A"
   - fatorRH[0][1] ← "+"
2. fatorRH[5] e {"A", "-"}
   - fatorRH[5][0] ← "O"
   - fatorRH[5][1] ← "-"
3. fatorRH[6] e {"AB", "+"}
   - fatorRH[6][0] ← "AB"
   - fatorRH[6][1] ← "+"
4. fatorRH[1] e {"A", "-"}
5. ...

Iniciação com cadeias de caracteres e literais compostos

Cadeias de caracteres

Cada caractere da cadeia, incluindo o caractere nulo ao final, inicia um elemento do vetor.

```
char vogais[] = "aeiou";
char let_ini[5] = "abcde";
char let_meio[5] = "klmnop";
char let_fim[5] = "xyz";
```

Iniciação com cadeias de caracteres e literais compostos

Cadeias de caracteres

Cada caractere da cadeia, incluindo o caractere nulo ao final, inicia um elemento do vetor.

```
char vogais[] = "aeiou";
char let_ini[5] = "abcde";
char let_meio[5] = "klmnop";
char let_fim[5] = "xyz";
```

Literais compostos

Os literais compostos podem iniciar vetores declarados como ponteiros.

```
int *primos = (int []){2, 3, 5, 7};
int *perfeitos = (int [4]){6, 28, 496, 8128};
```

Ponteiros e vetores

Na avaliação de expressões:

- Toda expressão do tipo *vetor de* $\langle T \rangle$ é convertida em uma expressão do tipo *ponteiro para* $\langle T \rangle$, cujo valor é um ponteiro apontando para o primeiro elemento do vetor.

  Exceções: operandos dos operadores `sizeof` *e* `&`*, e os literais cadeia de caracteres usados em expressões de iniciação.*

- As declarações de parâmetros de função do tipo *vetor de* $\langle T \rangle$ são ajustadas para declarações de parâmetros do tipo *ponteiro (possivelmente qualificado) para* $\langle T \rangle$.

Ponteiros e vetores

Exemplo

Considerando `vet` declarado como `char vet[3][2][4]`, então

Expressão	Tipo	Convertido em
`vet`	`char[3][2][4]`	`char (*)[2][4]`.
`vet[x]`	`char[2][4]`	`char (*)[4]`.
`vet[1][0]`	`char[4]`	`char *`.
`vet[x][y][w]`	`char`	Sem conversão a ponteiro.

Aritmética de ponteiros

Para uma variável `ptrA` do tipo *ponteiro para* $\langle T \rangle$ e um valor inteiro `qtd`:

1. A operação `ptrA + qtd` ou `qtd + ptrA` resulta no endereço que se obtém somando `qtd` $\times$ (tamanho de $\langle T \rangle$) a `ptrA`. Esse resultado é do tipo *ponteiro para* $\langle T \rangle$.

2. A operação `ptrA - qtd` resulta no endereço que se obtém subtraindo `qtd` $\times$ (tamanho de $\langle T \rangle$) de `ptrA`. Esse resultado é do tipo *ponteiro para* $\langle T \rangle$.
    - A operação `qtd - ptrA` não é definida.

Aritmética de ponteiros

Para variáveis `ptrA` e `ptrB` do tipo *ponteiro para* $\langle T \rangle$:

3. A operação `ptrA - ptrB` resulta no valor inteiro que corresponde à distância (orientada) entre os endereços `ptrA` e `ptrB` medida em termos do tamanho de $\langle T \rangle$: (endereço `ptrA` - endereço `ptrB`)/(tamanho de $\langle T \rangle$). Essa distância pode ser negativa e o valor obtido é do tipo `ptrdiff_t`.
   - Equivale à subtração dos índices que os ponteiros representam.
   - Definida para ponteiros que apontam para o mesmo tipo $\langle T \rangle$.

4. A operação `ptrA + ptrB` não é permitida.

Referenciando elementos dos vetores com ponteiros

- Para um vetor unidimensional `vet`, a referência
  - `vet[i]` corresponde a `*(vet + i)`.

Referenciando elementos dos vetores com ponteiros

- Para um vetor unidimensional `vet`, a referência
  - `vet[i]` corresponde a `*(vet + i)`.
- Para um vetor bidimensional `vet`, a referência
  - `vet[i][j]` corresponde a `*(*(vet + i) + j)`.

Referenciando elementos dos vetores com ponteiros

- Para um vetor unidimensional `vet`, a referência
  - `vet[i]` corresponde a `*(vet + i)`.
- Para um vetor bidimensional `vet`, a referência
  - `vet[i][j]` corresponde a `*(*(vet + i) + j)`.
- Para um vetor multidimensional `vet`, a referência
  - vet[i_1][i_2]...[i_n] corresponde a *(...*(*(vet + i_1) + i_2) + ... + i_n).

Referenciando elementos dos vetores com ponteiros

Exemplo

O programa ao lado lê e imprime, na ordem inversa à que foram digitados, um vetor de N números.

```
#include <stdio.h>
int main(void) {
  int N;
  do {
    scanf("%d", &N);
  } while ((N <= 0) || (N > 1000));
  int lval[N];
  for (int i = 0; i < N; i++) {
    scanf("%d", (lval + i));
  }
  for (int i = N - 1; i >= 0; i--) {
    if ((*(lval + i) % 2) == 0) {
      printf("%d ", *(lval + i));
    }
  }
  return 0;
}
```

Referenciando elementos dos vetores com ponteiros

Dois modos de usar ponteiros para referenciar os elementos de uma matriz:

Apenas ponteiros

```
void imp_vet(int L, int C, int m[L][C]) {
  for (int i = 0; i < L; i++) {
    for (int j = 0; j < C; j++) {
      printf("%d ", *(*(m + i) + j));
    }
    printf("\n");
  }
}
```

Ponteiros e índices

```
void imp_vet(int L, int C, int m[L][C]) {
  for (int i = 0; i < L; i++) {
    for (int j = 0; j < C; j++) {
      printf("%d ", (*(m + i))[j]);
    }
    printf("\n");
  }
}
```

Definindo tipos vetores

Na definição de tipos vetores com `typedef` usa-se a declaração do novo tipo como se fosse uma variável do tipo vetor que se desejar substituir:

`typedef int vA_t[10]`	Declara o tipo `vA_t` como sinônimo de `int [10]`.
`typedef char vB_t[3][6]`	Declara o tipo `vB_t` como sinônimo de `char [3][6]`.
`typedef float vC_t[][34]`	Declara o tipo `vC_t` como sinônimo de `float [][34]`.

Qualificando as variáveis do tipo vetor

`int const vet[x]`	A variável `vet` é declarada como um vetor de `int const`.
`int vet[const x]`	A variável `vet` é constante, declarada como um vetor de `int`.
`int vet[static 23]`	A variável `vet` é um vetor de `int`, com (*a expectativa de*) pelo menos 23 elementos.

Qualificando as variáveis do tipo vetor

`int const vet[x]`	A variável `vet` é declarada como um vetor de `int const`.
`int vet[const x]`	A variável `vet` é constante, declarada como um vetor de `int`.
`int vet[static 23]`	A variável `vet` é um vetor de `int`, com (*a expectativa de*) pelo menos 23 elementos.

Observação

A qualificação de variáveis do tipo vetor só pode ocorrer em protótipos e declaração de parâmetros.

Compatibilidade de vetores e ponteiros

Dois vetores são compatíveis se

- o tipo dos seus elementos são compatíveis,
- possuem as mesmas dimensões e
- os especificadores de tamanho, se existirem e forem constantes, têm o mesmo valor.

Compatibilidade de vetores e ponteiros

Dois vetores são compatíveis se

- o tipo dos seus elementos são compatíveis,
- possuem as mesmas dimensões e
- os especificadores de tamanho, se existirem e forem constantes, têm o mesmo valor.

Dois ponteiros são compatíveis se

- possuem os mesmos qualificadores e
- apontam para tipos compatíveis.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Estruturas e uniões

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Declaração de estruturas

As estruturas são declaradas através de variáveis do tipo estrutura que especifica os seus componentes.

Declaração de estruturas

As estruturas são declaradas através de variáveis do tipo estrutura que especifica os seus componentes.

Exemplo

Para o tipo estrutura ao lado as seguintes declarações são possíveis:

```
struct {
  int matr;
  float nota1;
  float nota2;
}
```

```
struct {
  int matr;
  float nota1;
  float nota2;
} aluno,
  aluno_regular;
```

```
struct r_aluno {
  int matr;
  float nota1;
  float nota2;
} aluno;
struct r_aluno
       aluno_regular,
       aluno_especial;
```

```
typedef struct {
  int matr;
  float nota1;
  float nota2;
} tp_r_aluno;
tp_r_aluno aluno,
   aluno_regular;
```

Operador de seleção direta

$\langle \mathit{OpSelecaoDireta} \rangle$ ::= $\langle \mathit{IdEstrutura} \rangle$ **.** $\langle \mathit{IdComponente} \rangle$

$\langle \mathit{IdEstrutura} \rangle$ ::= Identificador da estrutura, geralmente uma variável de tipo estrutura.

$\langle \mathit{IdComponente} \rangle$ ::= Identificador do componente.

Operador de seleção direta

$\langle OpSelecaoDireta \rangle$ ::= $\langle IdEstrutura \rangle$ **.** $\langle IdComponente \rangle$

$\langle IdEstrutura \rangle$ ::= Identificador da estrutura, geralmente uma variável de tipo estrutura.

$\langle IdComponente \rangle$ ::= Identificador do componente.

Exemplo

`aluno.nota1`	Componente `nota1` da estrutura armazenada em `aluno`.
`aluno.matr`	Componente `matr` da mesma estrutura.

Operador de seleção indireta

⟨*OpSelecaoIndireta*⟩ ::= ⟨*PtrEstrutura*⟩ -> ⟨*IdComponente*⟩

⟨*PtrEstrutura*⟩ ::= Variável do tipo ponteiro para estrutura.

⟨*IdComponente*⟩ ::= Identificador do componente.

Operador de seleção indireta

⟨*OpSelecaoIndireta*⟩ ::= ⟨*PtrEstrutura*⟩ -> ⟨*IdComponente*⟩

⟨*PtrEstrutura*⟩ ::= Variável do tipo ponteiro para estrutura.

⟨*IdComponente*⟩ ::= Identificador do componente.

Exemplo

`ptr_aluno->nota1` Componente `nota1` da estrutura apontada por `ptr_aluno`.

`ptr_aluno->matr` Componente `matr` da mesma estrutura.

Usando estruturas como valor de retorno e argumento

Exemplo

O programa ao lado lê os dados de um aluno, armazenando-os em uma estrutura e imprimindo-os em seguida.

```
#include <stdio.h>
struct r_aluno {
  int matr;
  float nota1;
  float nota2;
};
struct r_aluno ler_aluno(void);
void imp_aluno(struct r_aluno);
int main(void) {
  struct r_aluno aluno = ler_aluno();
  imp_aluno(aluno);
  return 0;
}
```

continua...

Usando estruturas como valor de retorno e argumento

Exemplo

...continuação

```
struct r_aluno ler_aluno(void) {
  struct r_aluno al;
  printf("Digite os dados do aluno\n");
  printf("Matricula: ");
  scanf("%d", &al.matr);
  printf("Primeira nota: ");
  scanf("%f", &al.nota1);
  printf("Segunda nota: ");
  scanf("%f", &al.nota2);
  return al;
}
void imp_aluno(struct r_aluno al) {
  printf("Matr: %d Notas: %5.2f %5.2f ",
           al.matr, al.nota1, al.nota2);
  printf("Media: %5.2f\n",
             (al.nota1 + al.nota2) / 2);
}
```

Ponteiros para estruturas

Exemplo

O programa ao lado modifica o exemplo anterior, usando um ponteiro para a estrutura aluno.

```
#include <stdio.h>
struct r_aluno {
  int matr;
  float nota1;
  float nota2;
};
void ler_aluno(struct r_aluno *);
void imp_aluno(struct r_aluno *);
int main(void) {
  struct r_aluno aluno;
  ler_aluno(&aluno);
  imp_aluno(&aluno);
  return 0;
}
```

continua...

Ponteiros para estruturas

Exemplo

...continuação

```
void ler_aluno(struct r_aluno *al) {
  printf("Digite os dados do aluno\n");
  printf("Matricula: ");
  scanf("%d", &al->matr);
  printf("Primeira nota: ");
  scanf("%f", &al->nota1);
  printf("Segunda nota: ");
  scanf("%f", &al->nota2);
}
void imp_aluno(struct r_aluno *al) {
  printf("Matr: %d Notas: %5.2f %5.2f ",
         al->matr, al->nota1, al->nota2);
  printf("Media: %5.2f\n",
            (al->nota1 + al->nota2) / 2);
}
```

Ponteiros para estruturas

- O conteúdo do endereço apontado por um ponteiro para estrutura é a própria estrutura.
- Se `al` é um ponteiro para uma estrutura, então `*al` é a própria estrutura apontada por `al`.

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

```
&(*al)

al->nota1
(*al).nota1
(&(*al))->nota1

&(*al).nota1
&((*al).nota1).
*al.nota1
```

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)` Endereço da estrutura armazenada no endereço apontado por `al`. Isto é, `&(*al)` = `al`.

```
al->nota1
(*al).nota1
(&(*al))->nota1

&(*al).nota1
&((*al).nota1).
*al.nota1
```

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)` Endereço da estrutura armazenada no endereço apontado por `al`. Isto é, `&(*al)` = `al`.

`al->nota1` Componente `nota1` da estrutura apontada por `al`.
`(*al).nota1`
`(&(*al))->nota1`

`&(*al).nota1`
`&((*al).nota1)`.
`*al.nota1`

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)`	Endereço da estrutura armazenada no endereço apontado por `al`. Isto é, `&(*al)` = `al`.
`al->nota1`	Componente `nota1` da estrutura apontada por `al`.
`(*al).nota1`	Componente `nota1` da estrutura `*al`.
`(&(*al))->nota1`	
`&(*al).nota1`	
`&((*al).nota1)`.	
`*al.nota1`	

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)`	Endereço da estrutura armazenada no endereço apontado por `al`. Isto é, `&(*al)` = `al`.
`al->nota1`	Componente `nota1` da estrutura apontada por `al`.
`(*al).nota1`	Componente `nota1` da estrutura `*al`.
`(&(*al))->nota1`	Componente `nota1` da estrutura cujo endereço é `&(*al)`.

`&(*al).nota1`

`&((*al).nota1).`

`*al.nota1`

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)`	Endereço da estrutura armazenada no endereço apontado por `al`. Isto é, `&(*al)` = `al`.
`al->nota1`	Componente `nota1` da estrutura apontada por `al`.
`(*al).nota1`	Componente `nota1` da estrutura `*al`.
`(&(*al))->nota1`	Componente `nota1` da estrutura cujo endereço é `&(*al)`.
`&(*al).nota1`	Endereço do componente `nota1` da estrutura `*al`.
`&((*al).nota1).`	
`*al.nota1`	

Ponteiros para estruturas

Exemplo

Considerando que `al` é um ponteiro para a estrutura `aluno` dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)`	Endereço da estrutura armazenada no endereço apontado por `al`. Isto é, `&(*al)` = `al`.
`al->nota1`	Componente `nota1` da estrutura apontada por `al`.
`(*al).nota1`	Componente `nota1` da estrutura `*al`.
`(&(*al))->nota1`	Componente `nota1` da estrutura cujo endereço é `&(*al)`.
`&(*al).nota1`	Endereço do componente `nota1` da estrutura `*al`.
`&((*al).nota1).`	Endereço do componente `nota1` da estrutura `*al`.
`*al.nota1`	

Ponteiros para estruturas

Exemplo

Considerando que al é um ponteiro para a estrutura aluno dos exemplos anteriores, o que representam as expressões a seguir?

`&(*al)`	Endereço da estrutura armazenada no endereço apontado por al. Isto é, &(*al) = al.
`al->nota1`	Componente nota1 da estrutura apontada por al.
`(*al).nota1`	Componente nota1 da estrutura *al.
`(&(*al))->nota1`	Componente nota1 da estrutura cujo endereço é &(*al).
`&(*al).nota1`	Endereço do componente nota1 da estrutura *al.
`&((*al).nota1).`	Endereço do componente nota1 da estrutura *al.
`*al.nota1`	Expressão inválida.

Estruturas com componente flexível

- O último componente de uma estrutura contendo mais de um componente nomeado pode ser de um tipo vetor incompleto.
  - *componente (vetor) flexível*.
- Antes do uso deve haver alocação explícita de espaço em memória para armazenar os elementos do vetor.
- O tamanho de um tipo estrutura com componente flexível é o mesmo que se o componente não existisse.

Estruturas com componente flexível

Exemplo

No programa ao lado a estrutura apontada pela variável `reg_a` é alocada com 4 elementos para o vetor `vendas_mes`.

```
#include <stdio.h>
#include <stdlib.h>
struct r1 {
  int matr;
  int vendas_mes[];
};
int main(void) {
  struct r1 *reg_a;
  reg_a = (struct r1 *)malloc(
      sizeof(struct r1) + sizeof(int[4]));
  for (int i = 0; i < 4; i++) {
    printf("%d\n", reg_a->vendas_mes[i]);
  }
  return 0;
}
```

Vetores de estruturas

Um vetor de estruturas é um vetor cujos elementos são de um tipo estrutura.

Exemplo

```
struct {
  int matr;
  float nota1;
  float nota2;
} v_alunos[20];
```

```
typedef struct {
  int matr;
  float nota1;
  float nota2;
} tp_r_aluno;
tp_r_aluno v_alunos[20];
```

Vetores de estruturas

Exemplo. O programa ao lado usa um vetor de estruturas para armazenar (e imprimir) os dados de dez alunos.

```
#include <stdio.h>
#define QTD_AL (10)
struct r_aluno {
  int matr;
  float nota1;
  float nota2;
};
void imp_alunos(struct r_aluno []);
int main(void) {
  struct r_aluno v_alunos[QTD_AL];
  printf("Digite os dados dos %d alunos\n",
          QTD_AL);
  for (int i = 0; i < QTD_AL; i++) {
    printf("Matricula %d: ", (i + 1));
    scanf("%d", &v_alunos[i].matr);
    printf("Primeira nota: ");
    scanf("%f", &v_alunos[i].nota1);
    printf("Segunda nota: ");
    scanf("%f", &v_alunos[i].nota2);
  }
```

continua...

Vetores de estruturas

Exemplo.

continuação...

```
  imp_alunos(v_alunos);
  return 0;
}
void imp_alunos(struct r_aluno al[]) {
  for (int i = 0; i < QTD_AL; i++) {
    printf("Matr: %d Notas: %5.2f %5.2f",
     al[i].matr, al[i].nota1, al[i].nota2);
    printf(" Media: %5.2f\n",
           (al[i].nota1 + al[i].nota2) / 2);
  }
}
```

Vetores de estruturas

Exemplo

Para o vetor `v_alunos` ao lado, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

```
v_alunos

*(v_alunos + 2)

&v_alunos[2]

v_alunos[2].matr

(*v_alunos).matr
```

Vetores de estruturas

Exemplo

Para o vetor `v_alunos` ao lado, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

`v_alunos` Ponteiro para a estrutura, apontando inicialmente para o primeiro elemento.

`*(v_alunos + 2)`

`&v_alunos[2]`

`v_alunos[2].matr`

`(*v_alunos).matr`

Vetores de estruturas

Exemplo

Para o vetor `v_alunos` ao lado, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

`v_alunos` Ponteiro para a estrutura, apontando inicialmente para o primeiro elemento.

`*(v_alunos + 2)` Estrutura armazenada no elemento apontado por `(v_alunos + 2)`.

`&v_alunos[2]`

`v_alunos[2].matr`

`(*v_alunos).matr`

Vetores de estruturas

Exemplo

Para o vetor `v_alunos` ao lado, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

`v_alunos`	Ponteiro para a estrutura, apontando inicialmente para o primeiro elemento.
`*(v_alunos + 2)`	Estrutura armazenada no elemento apontado por `(v_alunos + 2)`.
`&v_alunos[2]`	Endereço do terceiro elemento do vetor.
`v_alunos[2].matr`	
`(*v_alunos).matr`	

Vetores de estruturas

Exemplo

Para o vetor v_alunos ao lado, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

v_alunos	Ponteiro para a estrutura, apontando inicialmente para o primeiro elemento.
*(v_alunos + 2)	Estrutura armazenada no elemento apontado por (v_alunos + 2).
&v_alunos[2]	Endereço do terceiro elemento do vetor.
v_alunos[2].matr	Componente matr da estrutura armazenada no terceiro elemento do vetor.
(*v_alunos).matr	

Vetores de estruturas

Exemplo

Para o vetor v_alunos ao lado, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

v_alunos	Ponteiro para a estrutura, apontando inicialmente para o primeiro elemento.
*(v_alunos + 2)	Estrutura armazenada no elemento apontado por (v_alunos + 2).
&v_alunos[2]	Endereço do terceiro elemento do vetor.
v_alunos[2].matr	Componente matr da estrutura armazenada no terceiro elemento do vetor.
(*v_alunos).matr	Componente matr da estrutura armazenada no elemento apontado por v_alunos.

Vetores de estruturas

Exemplo

Se pa aponta para o segundo elemento do vetor ao lado, e sabendo que os operadores -> e . têm maior precedência que ++, o que representam as seguintes expressões?

```c
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

```c
pa++->matr

++pa->matr

++(pa->matr)

(++pa)->matr
```

Vetores de estruturas

Exemplo

Se pa aponta para o segundo elemento do vetor ao lado, e sabendo que os operadores -> e . têm maior precedência que ++, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

pa++->matr — Componente matr do segundo elemento. Ponteiro é incrementado.

++pa->matr

++(pa->matr)

(++pa)->matr

Vetores de estruturas

Exemplo

Se pa aponta para o segundo elemento do vetor ao lado, e sabendo que os operadores -> e . têm maior precedência que ++, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

`pa++->matr`	Componente matr do segundo elemento. Ponteiro é incrementado.
`++pa->matr`	Componente matr do segundo elemento. Componente é incrementado.
`++(pa->matr)`	
`(++pa)->matr`	

Vetores de estruturas

Exemplo

Se pa aponta para o segundo elemento do vetor ao lado, e sabendo que os operadores -> e . têm maior precedência que ++, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

pa++->matr	Componente matr do segundo elemento. Ponteiro é incrementado.
++pa->matr	Componente matr do segundo elemento. Componente é incrementado.
++(pa->matr)	Componente matr do segundo elemento. Componente é incrementado.
(++pa)->matr	

Vetores de estruturas

Exemplo

Se pa aponta para o segundo elemento do vetor ao lado, e sabendo que os operadores -> e . têm maior precedência que ++, o que representam as seguintes expressões?

```
struct r_aluno {
  int matr;
  float nota1, nota2;
} v_alunos[1000];
```

pa++->matr	Componente matr do segundo elemento. Ponteiro é incrementado.
++pa->matr	Componente matr do segundo elemento. Componente é incrementado.
++(pa->matr)	Componente matr do segundo elemento. Componente é incrementado.
(++pa)->matr	Componente matr do terceiro elemento. Ponteiro é incrementado.

Iniciando estruturas

- As variáveis do tipo estrutura podem ser iniciadas com uma relação de iniciação.

Exemplo

```
struct {
  int matr;
  float nota1;
  float nota2;
} aluno = {130518, 4.57, 8.33, 9.54},
  aluno_regular = {234, 7.65};
```

Iniciação seletiva e componentes agregados

- A iniciação seletiva se faz indicando o componente com a notação .⟨*nome_ componente*⟩
- Os componentes agregados podem ser especificados com chaves.

Exemplo

```
struct {
  int matr;
  struct {
    char sexo;
    int  rg;
  } sit;
  float nota1;
  float nota2;
} reg_a = {1111, 'm', 715, 8.54, 5.73},
  reg_b = {2222, 'f'},
  reg_c = {.sit = {'f', 32}, 215.3},
  reg_d = {3333, {'m', 234}, 86.44};
```

Iniciação com literais compostos

- As variáveis automáticas podem ser iniciadas com literais compostos.

Exemplo

```
struct r_aula {
  char topico[20];
  int ini;
  int fim;
} aula =
   (struct r_aula){"Programacao", 10, 12};
```

Declaração de uniões

As uniões são declaradas por meio do tipo união que especifica os seus componentes.

Declaração de uniões

As uniões são declaradas por meio do tipo união que especifica os seus componentes.

Exemplo

Para o tipo união ao lado as seguintes declarações são possíveis:

```
union {
  unsigned int regra;
  char norma;
}
```

```
union {
  unsigned int
           regra;
  char norma;
} id;
```

```
union doc_id {
  unsigned int
           regra;
  char norma;
};
union doc_id id;
```

```
typedef union {
  unsigned int
           regra;
  char norma;
} tp_doc_id;
tp_doc_id id;
```

Referenciando os componentes de uma união

Os componentes de uma união são referidos do mesmo modo que os componentes de uma estrutura:

- Por meio do operador de seleção direta
- Por meio do operador de seleção indireta

Usando os componentes apropriados

- O valor de uma união deve ser acessado através do componente utilizado para armazená-lo.
- O comportamento é indefinido se o valor de uma união é acessado com um componente diferente do componente usado para armazená-lo.

Usando os componentes apropriados

Exemplo

Para a união ao lado, após a atribuição `tst.num = 23`, as expressões a seguir têm o significado descrito:

```
union u {
  int num;
  char texto[20];
} tst;
```

`printf("%d", tst.num)` — Correto: imprime o conteúdo da variável `tst` como um valor do tipo `int`.

`printf("%s", tst.texto)` — Errado: imprime o conteúdo da variável `tst` como um valor do tipo `char [20]`.

`printf("%d", tst)` — Inapropriado: imprime o conteúdo da variável `tst`, convertendo-o do tipo `union u` em um valor do tipo `int`.

Usando os componentes apropriados

Exemplo

Em cada impressão do programa ao lado apenas um valor está corretamente referenciado.

```
#include <stdio.h>
union r {
  float a;
  double b;
  int c;
} tst;
int main(void) {
  tst.a = 34.5;
  printf("%g %g %d\n",tst.a, tst.b, tst.c);
  tst.b = 34.5;
  printf("%g %g %d\n",tst.a, tst.b, tst.c);
  tst.c = 34.5;
  printf("%g %g %d\n",tst.a, tst.b, tst.c);
  return 0;
}
```

O programa produz a seguinte saída:

```
34.5 5.47401e-315 1107951616
0 34.5 0
4.76441e-44 34.5 34
```

Uniões como componentes de estruturas

É comum usar uniões como componentes de estruturas, juntamente com um outro componente indicando como a união deve ser interpretada.

Exemplo

```
union doc_id {
  unsigned int
          regra;
  char    norma;
};
struct r_doc {
  char tipo;
  union doc_id id;
  char *resumo;
};
```

```
struct r_doc {
  char tipo;
  union {
    unsigned int
            regra;
    char    norma;
  } id;
  char *resumo;
};
```

```
typedef union {
  unsigned int
          regra;
  char    norma;
} tp_id;
struct r_doc {
  char tipo;
  tp_id id;
  char *resumo;
};
```

Uniões como componentes de estruturas

Exemplo. O programa ao lado lê o tipo e a identificação de um documento. Existem dois formatos para a identificação, variando de acordo com o tipo. Ambos são armazenados em uma mesma variável do tipo união.

continua...

```
#include <stdio.h>
typedef union {
  unsigned int regra;
  char norma;
} tp_id;
struct r_doc {
  char tipo;
  tp_id id;
  char *resumo;
};
void obtem_id(tp_id *, char);
void imp_doc(struct r_doc *);
void limpa_linha(void);
int main(void) {
  struct r_doc doc;
  printf("Digite o tipo de documento: ");
  doc.tipo = getchar();
  limpa_linha();
  obtem_id(&doc.id, doc.tipo);
  imp_doc(&doc);
  return 0;
}
```

Uniões como componentes de estruturas

...continuação,

```
void obtem_id(tp_id *id, char tipo) {
  if (tipo == 'r') {
    printf("id regra (numerico): ");
    scanf("%u", &id->regra);
  } else {
    if (tipo == 'n') {
      printf("id norma (char): ");
      scanf("%c", &id->norma);
    }
  }
}
```

continua...

Uniões como componentes de estruturas

...continuação.

```
void imp_doc(struct r_doc *d) {
  printf("tipo: %c id: ", d->tipo);
  if (d->tipo == 'r') {
    printf("%u\n", d->id.regra);
  } else {
    if (d->tipo == 'n') {
      printf("%c\n", d->id.norma);
    } else {
      printf("tipo invalido\n");
    }
  }
}
void limpa_linha() {
  while (getchar() != '\n') { };
}
```

Iniciando uniões

As uniões são explicitamente iniciadas com o valor correspondente ao seu primeiro componente especificado entre chaves, podendo haver a iniciação seletiva, indicando-se um componente específico da união.

Iniciando uniões

As uniões são explicitamente iniciadas com o valor correspondente ao seu primeiro componente especificado entre chaves, podendo haver a iniciação seletiva, indicando-se um componente específico da união.

Exemplo

Para a declaração de tipo ao lado,

```
union doc_id {
  unsigned int regra;
  char norma;
};
```

Iniciação	Corresponde a
`union doc_id id = {1234};`	`id.regra = 1234;`
`union doc_id id = {’t’};`	`id.regra = ’t’;`
`union doc_id id = {.norma = ’t’};`	`id.norma = ’t’;`

Campos de bits

- Define componentes de tipo inteiro com uma quantidade limitada de bits.
- O tipo de um campo de bits deve ser
  - `_Bool`,
  - `signed int`,
  - `unsigned int`, ou
  - algum outro tipo dependente da implementação.
- O tipo `int` será implementado como `signed int` ou `unsigned int`, dependendo da implementação.

Campos de bits

- Define componentes de tipo inteiro com uma quantidade limitada de bits.
- O tipo de um campo de bits deve ser
  - `_Bool`,
  - `signed int`,
  - `unsigned int`, ou
  - algum outro tipo dependente da implementação.
- O tipo `int` será implementado como `signed int` ou `unsigned int`, dependendo da implementação.

Exemplo

A estrutura ao lado usa dois campos de bits.

```
struct r_arq {
  char id[9];
  _Bool grv: 1;
  unsigned int perm: 3;
}
```

Campos de bits

São altamente dependentes da implementação:

- Não podem ser acessados com o operador de endereço (`&`).
- Podem ser não-nomeados.
- Os campos não-nomeados podem ser especificados com zero bits.
  - Um campo de bits pode ser alocado na mesma unidade de armazenamento que o campo precedente, ou não.
  - O uso de um campo com 0 bits faz com que o próximo não seja alocado na mesma unidade que o campo precedente.

Compatibilidade de estruturas e uniões

Duas estruturas ou uniões (declaradas em diferentes unidades de compilação) são compatíveis se

1. possuem a mesma etiqueta.
2. Adicionalmente, se os tipos são completos, deve existir uma correspondência entre os seus componentes tal que os componentes correspondentes devem
   - ser declarados na mesma ordem (apenas para estruturas);
   - ter tipos compatíveis;
   - ter nomes iguais (se forem nomeados); e
   - ser do mesmo tamanho (se forem campos de bits).

Compatibilidade de estruturas e uniões

Exemplo

A compilação do seguinte programa gera um executável com comportamento indefinido.

prgA.c

```
#include <stdio.h>
void fun (void);
struct s1 {
int a;
int b;
};
struct s1
   var1 = {33, 44};
struct s1 var2;
int main(void) {
var2 = var1 ;
fun();
return 0;
}
```

prgB.c

```
#include <stdio.h>
struct s1 {
int a, b;
char c;
};
struct s1 var2 = {111 , 222 , 's'};
void fun(void) {
printf ("%d %d %c\n", var2.a, var2.b,
                              var2.c);
}
```

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

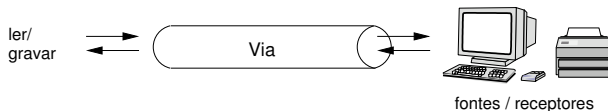
Entrada e saída: teclado e monitor de vídeo

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Vias lógicas de comunicação

Representam um canal de comunicação entre uma fonte ou repositório de dados e um programa.

Classificação das vias de comunicação

As vias lógicas de comunicação são classificadas quanto

- Tipo
- Modo de operação
- Acesso
- Orientação

Tipo

Vias de texto. Implementadas como sequências de caracteres organizados em linha.

Vias binárias. Implementadas como sequências de caracteres organizados de modo a representar de forma direta os valores dos tipos básicos.

Modo de operação

Vias de entrada. Permitem operações de leitura.

Vias de saída. Permitem operações de gravação.

Vias de entrada e saída. Permitem operações de leitura e gravação.

Acesso

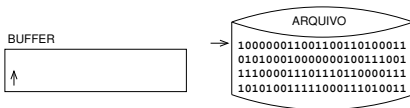
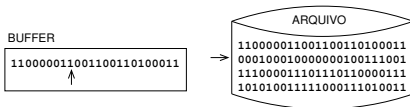
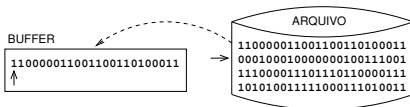
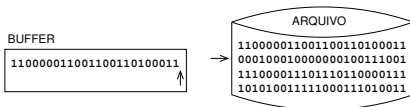
Vias sequenciais ou de acesso sequencial. O cursor de posição se move apenas em uma direção, do início para o fim da via.

Vias randômicas ou de acesso randômico. O cursor de posição pode se mover em ambas as direções, para o início ou para o fim da via.

Orientação

Via orientada a byte. Cada caractere é associado a um byte (e vice-versa).

Via orientada a caractere multibyte. Cada caractere é associado à sequência de bytes que o representa (e vice-versa).

Área de armazenamento temporário

Lê 1 byte

Lê 2 bytes

Processo de leitura

1. Declarar uma variável que possa armazenar a identificação de uma via de comunicação.
2. Criar a via de comunicação e associá-la à fonte de dados que se deseja ler.
3. Realizar as operações de leitura usando a (variável que contém a identificação da) via de comunicação.
4. Fechar a via de comunicação após as leituras do programa.

Processo de gravação

1. Declarar uma variável que possa armazenar a identificação de uma via de comunicação.
2. Criar a via de comunicação e associá-la ao repositório de dados no qual se deseja gravar.
3. Realizar as operações de gravação usando a (variável que contém a identificação da) via de comunicação.
4. Fechar a via de comunicação após as gravações do programa.

Biblioteca de entrada e saída

`stdio.h`

- `FILE`
- `EOF`
- `stdin`
- `stdout`
- `stderr`

Lendo dados do teclado — caracteres

`int getchar(void)`

Lê um caractere da entrada padrão como um valor do tipo `unsigned char`.

Valor de retorno. O código do caractere lido como um valor do tipo `int`.

Lendo dados do teclado — caracteres

Exemplo

```
#include <stdio.h>
int main(void) {
  int c;
  printf("Digite algo: ");
  c = getchar();
  printf("Voce digitou: ");
  printf("%d (codigo de %c)\n", c, c);
  return 0;
}
```

Lendo dados do teclado — linhas

```
char *gets(char *linha)
```

Lê da entrada padrão todos os caracteres digitados até que ocorra um fim de arquivo ou que seja digitado o caractere de fim de linha (tecla de retorno). Os caracteres lidos são armazenados na cadeia de caracteres apontada por `linha`. O caractere de fim de linha que finaliza a leitura é lido mas não é armazenado, e o caractere nulo é inserido após o último caractere armazenado em `linha`.

Valor de retorno. Um ponteiro para a cadeia `linha` ou o ponteiro nulo se ocorrer um erro de leitura ou se ocorrer o fim de arquivo e nenhum caractere houver sido digitado.

Esta função é obsoleta, devendo ser usada a função `get_s` em seu lugar, se disponível.

Lendo dados do teclado — linhas

Exemplo

```
#include <stdio.h>
#include <string.h>
int main(void) {
  int qtd = 0;
  char linha[255];
  printf("Digite uma linha: ");
  gets(linha);
  for (int i = 0; i < strlen(linha); i++)
  {
    if (linha[i] == 'a') {
      qtd++;
    }
  }
  printf("A linha digitada contem ");
  printf("%d letras 'a'", qtd);
  return 0;
}
```

Lendo dados do teclado — tipos básicos

```
int scanf(const char * restrict formato, ...)
```

Lê valores da entrada padrão, convertendo-os em valores dos tipos básicos, segundo as diretivas de conversão presentes na cadeia apontada por `formato`, e armazenando-os nas variáveis apontadas pelos argumentos da parte variável. Os argumentos da parte variável devem ser ponteiros para as variáveis que receberão os valores convertidos.

Valor de retorno. A quantidade de valores atribuídos ou `EOF`, se ocorre um erro de leitura antes de qualquer conversão. A função retorna ao final do processamento de todas as diretivas ou tão logo a aplicação de alguma diretiva falhe. Assim, o valor de retorno pode ser menor que a quantidade de diretivas, inclusive zero.

Diretivas — conversões inteiras

	Valor lido	Argumento correspondente
`d`	inteiro decimal	ponteiro para `int`
`i`	inteiro, interpretado como decimal, octal (antecedido de `0`) ou hexadecimal (antecedido de `0x` ou `0X`)	ponteiro para `int`
`u`	inteiro decimal	ponteiro para `unsigned int`
`o`	inteiro octal (com ou sem o prefixo `0`)	ponteiro para `unsigned int`
`x`, `X`	inteiro hexadecimal, (com ou sem `0x`, `0X`)	ponteiro para `unsigned int`

Diretivas — conversões inteiras

Exemplo

```
void le_exem(int *numA, int *numB) {
  unsigned int numC, numD, numE;
  scanf("%d", numA);
  scanf("%i", numB);
  scanf("%o %u", &numC, &numD);
  scanf("%x", &numE);
  /* codigo omitido */
}
```

Diretivas — conversões reais

	Valor lido	Argumento correspondente
a, A	número real, infinito ou NAN	ponteiro para float
e, E	pode ser expresso como hexadecimal,	
f, F	se precedido do prefixo 0x ou 0X	
g, G		

Diretivas — conversões reais

Exemplo

```
float numA, numB, numC;
scanf("%a %f", &numA, &numB);
scanf("%g", &numC);
/* codigo omitido */
```

Diretivas — conversões de caracteres

	Valor lido	Argumento correspondente
c	caractere	ponteiro para char
s	sequência de caracteres diferentes de espaço. O caractere nulo é inserido no fim da cadeia	ponteiro para char []
[	sequência de caracteres pertencentes ao conjunto especificado entre colchetes. O caractere nulo é inserido no fim da cadeia	ponteiro para char []

Diretivas — conversões de caracteres

Exemplo

```
char letra;
char nome[20];
scanf("%c", &letra);
scanf("%s", nome);
```

Exemplo

```
char nome1[20], nome2[20], nome3[20];
scanf("%[a-cm-o ]", nome1);
scanf("%[^a-ckls-u]", nome2);
scanf("%[][a-c ]", nome3);
```

Diretivas — miscelânea

	Valor lido	Argumento correspondente
`p`	endereço	ponteiro para ponteiro `void`
`n`	nenhum. Armazena no argumento a quantidade de caracteres lidos	ponteiro para `int`
`%`	deve ser %	não possui

Diretivas — miscelânea

Exemplo

```
void *end;
int num, qtd;
float val;
scanf("%p", &end);
scanf("%d%n%f", &num, &qtd, &val);
```

Modificadores de tipo

Modificador	Especificador de conversão	Argumento correspondente
hh	d, i ou n	signed char *
	o, u, x ou X	unsigned char *
h	d, i ou n	short int *
	o, u, x ou X	unsigned short int *
l	d, i ou n	long int *
	o, u, x ou X	unsigned long int *
	a, A, f, F, g, G, e ou E	double *
	c, s ou [	wchar_t *
ll	d, i ou n	long long int *
	o, u, x ou X	unsigned long long int *

Modificadores de tipo

Modificador	Especificador de conversão	Argumento correspondente
j	d, i ou n o, u, x ou X	intmax_t * uintmax_t *
z	d, i ou n o, u, x, X	(size_t sinalizado) * size_t *
t	d, i ou n o, u, x, X	ptrdiff * (ptrdiff não sinalizado) *
L	a, A, f, F, g, G, e ou E	long double *

Interrompendo a leitura

Tamanho máximo do campo

O indicador de tamanho interrompe a leitura quando o número de caracteres lidos é igual ao especificado por ele.

Interrompendo a leitura

Exemplo

O trecho de programa ao lado produz as seguintes atribuições:

```
int n1;
long int n2;
float n3;
double n4;
scanf("%d %3ld %4f %5lf",
                    &n1, &n2, &n3, &n4);
```

		Atribuições			
Caso	Digitação	n1	n2	n3	n4
1)	137 23 8.3 95.1	137	23	8,3	95,1
2)	213356 4792 9.7	213.356	479	2,0	9,7
3)	12 76187.4273951 8.6	12	761	87,4	27.395,0
4)	23 4 0x2345.4 44	23	4	35,0	45,4

Suprimindo a atribuição

- O uso do asterisco indica que o valor lido não deve ser atribuído.
- A leitura e a conversão são realizadas, apenas a atribuição é suprimida.

Exemplo

A seguinte função lê 3 dígitos (que podem compor um valor do tipo `long int`, desprezando os dígitos lidos. Em seguida, um valor do tipo `int` é lido e armazenado em `val`:

```
scanf("%*3ld %d", &val);
```

Suprimindo caracteres remanescentes

Os caracteres remanescentes na área de armazenamento temporário, após a leitura de um valor do teclado, podem ser suprimidos com o seguinte código:

```
void limpa_linha(void) {
  scanf("%*[^\n]");
  scanf("%*c");
}
```

Gravando dados — caracteres

```
int putchar(int c)
```

Grava na saída padrão o caractere `c` convertido em um valor do tipo `unsigned char`.

Valor de retorno. O código decimal do caractere gravado, como um valor do tipo `int`, ou `EOF`, em caso de falha.

Gravando dados — caracteres

Exemplo

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
bool vogal(char);
int main(void) {
  char linha[31];
  printf("Digite algo: ");
  scanf("%30[^\n]", linha);
  for (int i = strlen(linha) - 1;
                            i >= 0; i--) {
    if (vogal(linha[i])) {
      putchar(linha[i]);
    }
  }
  return 0;
}
```

continua...

Gravando dados — caracteres

Exemplo

...continuação.

```
bool vogal(char c) {
  switch (c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': return true;
    default:  return false;
  }
}
```

Gravando dados — linhas

```
int puts(const char * restrict linha)
```

Grava na saída padrão a cadeia de caracteres apontada por seu argumento. A cadeia `linha` deve ser terminada por um caractere nulo que, entretanto, não é gravado. Por outro lado, um caractere de fim de linha é sempre gravado após a gravação dos caracteres de `linha`.

Valor de retorno. Um valor não-negativo ou `EOF`, em caso de erro.

Gravando dados — linhas

Exemplo

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
bool vogal(char);
int main(void) {
  int j = 0;
  char orig[31];
  char dest[31];
  printf("Digite algo: ");
  scanf("%30[^\n]", orig);
  for (int i = 0; i < strlen(orig); i++) {
    if (!vogal(orig[i])) {
      dest[j++] = orig[i];
    }
  }
  dest[j] = '\0';
  puts(dest);
  return 0;
}
```

Gravando dados — tipos básicos

```
int printf(const char * restrict formato, ...)
```

Grava na saída padrão sequências de caracteres que representam valores dos tipos básicos armazenados nas variáveis apontadas pelos argumentos da parte variável. A formatação de um valor de um tipo básico como uma sequência de caracteres que o representa dá-se através das diretivas de formatação presentes na cadeia apontada por `formato`.

Valor de retorno. A quantidade de caracteres gravados ou um valor negativo, em caso de erro de entrada e saída ou de formato.

Diretivas — formatações inteiras

Diretiva	Argumento	Valor impresso
`d`, `i`	`int`	inteiro no formato decimal
`u`	`unsigned int`	inteiro decimal não sinalizado
`o`	`unsigned int`	inteiro octal não sinalizado
`x`, `X`	`unsigned int`	inteiro hexadecimal não sinalizado

Diretivas — formatações inteiras

Exemplo

```
#include <stdio.h>
int main(void) {
  int a = 155, b = -155;
  printf("%d %i %o %x\n", a, a, a, a);
  printf("%d %i %o %x\n", b, b, b, b);
  return 0;
}
```

Diretivas — formatações reais

Diretiva	Argumento	Valor impresso
`f`, `F`	`double`	no formato: [-]⟨*ParteInteira*⟩ **.** ⟨*ParteFracionária*⟩
`g`, `G`	`double`	no formato da diretiva `f` ou `e`, dependendo da precisão
`e`, `E`	`double`	no formato científico: [-]⟨*ParteInteira*⟩ **.** ⟨*ParteFracionária*⟩ [**e**] ⟨*Expoente*⟩
`a`, `A`	`double`	no formato hexadecimal: [-]**0x**⟨*ParteInteira*⟩ **.** ⟨*ParteFracionária*⟩ **p** ⟨*Expoente*⟩

Diretivas — formatações reais

Exemplo

```
#include <stdio.h>
#include <math.h>
int main(void) {
  float  a = 639.87f; double b = 639.87;
  printf("%f %e\n", a, a);
  printf("%f %E\n", b, b);
  double c = 0.000528, d = 0.0000528;
  printf("%f %e %g\n", c, c, c);
  printf("%f %E %g\n", d, d, d);
  b = 6.12;
  printf("%a\n", b);
  return 0;
}
```

Diretivas — formatações de caracteres

Diretiva	Argumento	Valor impresso
c	int	caractere, após conversão em unsigned int
s	char *	cadeia apontada pelo argumento

Diretivas — formatações de caracteres

Exemplo

```
#include <stdio.h>
int main(void) {
  char letra = 'o';
  char verso[] = "Carregado de mim";
  printf("%s and%c n%c mund%c\n", verso,
                      'o', letra, letra);
  return 0;
}
```

Diretivas — miscelânea

Diretiva	Argumento	Valor impresso
`p`	`void *`	representação do endereço apontado pelo argumento
`n`	`int *`	quantidade de caracteres impressos até o momento
`%`		caractere '%'

Diretivas — miscelânea

Exemplo

```
#include <stdio.h>
int main(void) {
  double a = 12.346;
  int q1, q2;
  printf("Endereco a= %p\n", (void *)&a);
  printf("a = %n%f%n; impresso ",
                              &q1, a, &q2);
  printf("com %d caracteres\n", q2 - q1);
  return 0;
}
```

Modificadores de tipo

Modificador	Especificador de conversão	Argumento correspondente
hh	d ou i	signed char
	o, u, x ou X	unsigned char
	n	signed char *
h	d ou i	short int
	o, u, x ou X	unsigned short int
	n	short int *
l	d ou i	long int
	o, u, x ou X	unsigned long int
	c	wint_t
	s	wchar_t *
	n	long int *
	a, A, f, F, g, G, e ou E	sem efeito

Modificadores de tipo

Modificador	Especificador de conversão	Argumento correspondente
ll	d ou i	long long int
	o, u, x ou X	unsigned long long int
	n	long long int *
j	d ou i	intmax_t
	o, u, x ou X	uintmax_t
	n	intmax_t *
z	d ou i	size_t sinalizado
	o, u, x, X	size_t
	n	(intmax_t sinalizado) *
t	d ou i	ptrdiff_t
	o, u, x, X	ptrdiff_t não sinalizado
	n	ptrdiff_t *
L	a, A, f, F, g, G, e ou E	long double

Modificadores de formato

Mod.	Efeito
#	Torna explícita a notação utilizada: o: sempre inclui 0 inicial x, X: usa prefixos 0x (ou 0X) a, A, f, F, e, E, g, G: sempre inclui o ponto decimal g, G: não remove os zeros finais da parte fracionária
-	alinhamento à esquerda
+	sempre imprime o sinal (+ ou -)
⟨*espaço*⟩	imprime um espaço à esquerda se o valor é positivo ou não possui caracteres
0	imprime zeros em vez de espaços para fazer com que o valor impresso tenha o tamanho mínimo

Tamanho mínimo

- Define a quantidade mínima de caracteres que deve ser usada para representar o valor convertido.
- Não há truncamento:
  - Se o valor possui um número menor de caracteres, espaços ou zeros são usados para atingir o tamanho especificado.
  - Se o valor formatado possui um número maior de caracteres, ele é impresso com todos os seus caracteres
- Se o tamanho mínimo é especificado por um asterisco,
  - deve haver um argumento de tamanho, do tipo `int`, imediatamente antes do argumento que corresponde à diretiva, especificando o tamanho mínimo.

Precisão

- Especificada por um ponto seguido opcionalmente de um valor inteiro não-negativo ou asterisco: . [⟨*VlrPrecisão*⟩ | *]
- O significado da precisão depende da diretiva de formatação:

Diretiva	Significado
`d`, `i`, `o`, `u`, `x` e `X`	Número mínimo de dígitos do valor formatado
`a`, `A`, `e`, `E`, `f` e `F`	Número de dígitos após o ponto decimal
`g` e `G`	Idêntica às demais formatações reais, exceto que os zeros finais da parte fracionária são removidos
`s`	Número máximo de caracteres que devem ser impressos

- Se a precisão é especificada por um asterisco, então deve haver um argumento do tipo `int` contendo o valor da precisão.

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	
`printf("|%+4Lf|", val)`	
`printf("|%13Lf|", -val)`	
`printf("|%013Lf|", -val)`	
`printf("|%13.Lf|", val)`	
`printf("|%13.2Lf|", val)`	
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	|450.300000|
`printf("|%+4Lf|", val)`	
`printf("|%13Lf|", -val)`	
`printf("|%013Lf|", -val)`	
`printf("|%13.Lf|", val)`	
`printf("|%13.2Lf|", val)`	
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	|450.300000|
`printf("|%+4Lf|", val)`	|+450.300000|
`printf("|%13Lf|", -val)`	
`printf("|%013Lf|", -val)`	
`printf("|%13.Lf|", val)`	
`printf("|%13.2Lf|", val)`	
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	
`printf("|%13.Lf|", val)`	
`printf("|%13.2Lf|", val)`	
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	`|-00450.300000|`
`printf("|%13.Lf|", val)`	
`printf("|%13.2Lf|", val)`	
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	`|-00450.300000|`
`printf("|%13.Lf|", val)`	`|          450|`
`printf("|%13.2Lf|", val)`	
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	`|-00450.300000|`
`printf("|%13.Lf|", val)`	`|          450|`
`printf("|%13.2Lf|", val)`	`|       450.30|`
`printf("|%13Lg|", val)`	
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	`|-00450.300000|`
`printf("|%13.Lf|", val)`	`|          450|`
`printf("|%13.2Lf|", val)`	`|       450.30|`
`printf("|%13Lg|", val)`	`|        450.3|`
`printf("|%13.Lg|", val)`	
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	`|-00450.300000|`
`printf("|%13.Lf|", val)`	`|          450|`
`printf("|%13.2Lf|", val)`	`|       450.30|`
`printf("|%13Lg|", val)`	`|        450.3|`
`printf("|%13.Lg|", val)`	`|        5e+02|`
`printf("|%13.2Lg|", val)`	

Exemplos

Se `val` é declarada como `long double val = 450.3L;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%Lf|", val)`	`|450.300000|`
`printf("|%+4Lf|", val)`	`|+450.300000|`
`printf("|%13Lf|", -val)`	`|  -450.300000|`
`printf("|%013Lf|", -val)`	`|-00450.300000|`
`printf("|%13.Lf|", val)`	`|          450|`
`printf("|%13.2Lf|", val)`	`|       450.30|`
`printf("|%13Lg|", val)`	`|        450.3|`
`printf("|%13.Lg|", val)`	`|        5e+02|`
`printf("|%13.2Lg|", val)`	`|      4.5e+02|`

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	
`printf("|%13.f|", val)`	
`printf("|%13.2f|", val)`	
`printf("|%13g|", val)`	
`printf("|%#13g|", val)`	
`printf("|%13.g|", val)`	
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	
`printf("|%13.2f|", val)`	
`printf("|%13g|", val)`	
`printf("|%#13g|", val)`	
`printf("|%13.g|", val)`	
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	
`printf("|%13g|", val)`	
`printf("|%#13g|", val)`	
`printf("|%13.g|", val)`	
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	
`printf("|%#13g|", val)`	
`printf("|%13.g|", val)`	
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	`|          405|`
`printf("|%#13g|", val)`	
`printf("|%13.g|", val)`	
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	`|          405|`
`printf("|%#13g|", val)`	`|      405.000|`
`printf("|%13.g|", val)`	
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	`|          405|`
`printf("|%#13g|", val)`	`|      405.000|`
`printf("|%13.g|", val)`	`|        4e+02|`
`printf("|%#13.g|", val)`	
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	`|          405|`
`printf("|%#13g|", val)`	`|      405.000|`
`printf("|%13.g|", val)`	`|        4e+02|`
`printf("|%#13.g|", val)`	`|       4.e+02|`
`printf("|%13.2g|", val)`	
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	`|          405|`
`printf("|%#13g|", val)`	`|      405.000|`
`printf("|%13.g|", val)`	`|        4e+02|`
`printf("|%#13.g|", val)`	`|       4.e+02|`
`printf("|%13.2g|", val)`	`|        4e+02|`
`printf("|%#13.2g|", val)`	

Exemplos

Se `val` é declarada como `double val = 405.0;` qual é o resultado das seguintes impressões?

Função	Resultado
`printf("|%13f|", val)`	`|   405.000000|`
`printf("|%13.f|", val)`	`|          405|`
`printf("|%13.2f|", val)`	`|       405.00|`
`printf("|%13g|", val)`	`|          405|`
`printf("|%#13g|", val)`	`|      405.000|`
`printf("|%13.g|", val)`	`|        4e+02|`
`printf("|%#13.g|", val)`	`|       4.e+02|`
`printf("|%13.2g|", val)`	`|        4e+02|`
`printf("|%#13.2g|", val)`	`|      4.0e+02|`

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Entrada e saída: arquivos

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Classificação

Tipo

Arquivos textos. São implementados como sequências de caracteres organizados em linha.

Arquivos binários. São implementados como sequências de caracteres representando valores dos tipos básicos.

Classificação

Tipo

Arquivos textos. São implementados como sequências de caracteres organizados em linha.

Arquivos binários. São implementados como sequências de caracteres representando valores dos tipos básicos.

Modo de operação

Leitura. Operações de leitura.

Gravação ou adição. Operações de gravação. No modo de adição as gravações ocorrem apenas no fim do arquivo.

Atualização. Operações de leitura e gravação.

Identificação

A identificação externa de um arquivo é determinada

1. pelo caminho que indica sua localização no ambiente de execução,
2. seguido do seu nome.

Identificação

A identificação externa de um arquivo é determinada

1. pelo caminho que indica sua localização no ambiente de execução,
2. seguido do seu nome.

Exemplo

Arquivo de nome `JosefaFlor.txt`, armazenado no subdiretório `cartas`, do diretório `exemplo`.

`exemplo/cartas/JosefaFlor.txt`

Organização dos dados

- Um registro é um conjunto de informações relacionadas entre si
  - Formato fixo
  - Formato variável
- Um campo é uma informação distinta de um registro

	arquivo A	arquivo B	
Registro:	`matricula`	`aluno`	
Campos:	`matricula`	`matricula`	`nome`

Utilização de arquivos

1. Declarar uma variável para armazenar a identificação de uma via de comunicação.
2. Criar a via de comunicação e associá-la à fonte (repositório) de dados que se quer utilizar.
3. Realizar as operações de leitura ou gravação usando a (variável que contém a identificação da) via de comunicação.
4. Fechar a via de comunicação após sua utilização.

Utilização de arquivos

1. Declarar uma variável para armazenar a identificação de uma via de comunicação.

```
FILE *arq;
```

2. Criar a via de comunicação e associá-la à fonte (repositório) de dados que se quer utilizar.

```
arq = fopen("nome.arq", "r");
```

3. Realizar as operações de leitura ou gravação usando a (variável que contém a identificação da) via de comunicação.

```
fgetc(arq);
```

4. Fechar a via de comunicação após sua utilização.

```
fclose(arq);
```

Abrindo arquivos

```
FILE *fopen(const char * restrict nome_arq,
                                  const char * restrict modo)
```

Abre o arquivo cujo nome é apontado por `nome_arq`, no modo de operação indicado por `modo`.

Valor de retorno. Um ponteiro para a via associada ao arquivo ou o ponteiro nulo, em caso de falha.

Abrindo arquivos

O modo de operação indica o tipo da via associada ao arquivo:

Arquivo texto	Arquivo binário	Operação
`r`	`rb`	Abre um arquivo para leitura.
`w`	`wb`	Cria (ou trunca) um arquivo para gravação.
`wx`	`wbx`	Cria um arquivo para gravação.
`a`	`ab`	Abre ou cria um arquivo para adição (gravação a partir do final).
`r+`	`rb+` ou `r+b`	Abre um arquivo para atualização (leitura ou gravação).
`w+`	`wb+` ou `w+b`	Cria (ou trunca) um arquivo p/atualização.
`w+x`	`wb+x` ou `w+bx`	Cria um arquivo para atualização.
`a+`	`ab+` ou `a+b`	Abre ou cria um arquivo para atualização (gravação a partir do final).

Fechando arquivos

```
int fclose(FILE *arq)
```

Fecha o arquivo associado à via apontada por `arq`, após gravar todos os dados ainda não gravados, se a via é de saída. Se a via é de entrada, os dados ainda não lidos que estão na área de armazenamento temporário do arquivo são desprezados. Após o fechamento, a via é desassociada do arquivo e todos as áreas de armazenamento temporário são liberadas (caso tenham sido automaticamente alocadas).

Valor de retorno. 0, se a operação for bem sucedida, ou `EOF`, em caso de falha.

Abrindo e Fechando arquivos

Exemplo

```
#include <stdio.h>
#define QTD (30)
char *ler_linha(char *);
int main(void) {
  FILE *arq;
  char nome[QTD];
```

continua...

Abrindo e Fechando arquivos

Exemplo

...continuação

```
  do {
    printf("Nome do arquivo ");
    printf("(<Enter> p/terminar): ");
    if (scanf("%29[^\n]", nome) == 0) {
      return 0; /* Fim de programa */
    }
    scanf("%*[^\n]"); scanf("%*c");
    arq = fopen(nome, "r");
    if (arq == NULL) {
      printf("Erro ao abrir |%s|\n", nome);
    }
  } while (arq == NULL);
  printf("Arquivo existe\n");
  fclose(arq);
  return 0;
}
```

Detectando o fim de arquivo

- Existe um indicador de fim de arquivo associado à via que o acessa.
- A função `feof` verifica o estado do indicador de fim de arquivo.
- Entretanto, o indicador é ativado apenas quando ocorre uma tentativa de leitura após o último dado gravado.

Detectando o fim de arquivo

- Existe um indicador de fim de arquivo associado à via que o acessa.
- A função feof verifica o estado do indicador de fim de arquivo.
- Entretanto, o indicador é ativado apenas quando ocorre uma tentativa de leitura após o último dado gravado.

Este esquema não pode ser utilizado:

```
while (feof( arquivo ) != valor que indica fim de arquivo) {
       /* lê e processa o registro lido */
}
```

Detectando o fim de arquivo

- Existe um indicador de fim de arquivo associado à via que o acessa.
- A função `feof` verifica o estado do indicador de fim de arquivo.
- Entretanto, o indicador é ativado apenas quando ocorre uma tentativa de leitura após o último dado gravado.

Este esquema não pode ser utilizado:

```
while (feof( arquivo ) != valor que indica fim de arquivo) {
      /* lê e processa o registro lido */
}
```

Este esquema deve ser usado com cautela:

```
if ((dado = função_leitura()) != valor que indica fim de arquivo) {
      /* dado foi lido e não é fim de arquivo */
}
```

Lendo caracteres

```
int fgetc(FILE *arq)
```

Obtém o próximo caractere da via apontada por `arq`, como um valor do tipo `unsigned char`; dessa forma, garante-se que todo caractere válido será positivo.

Valor de retorno. O código do caractere lido convertido em um valor do tipo `int`, se for bem sucedida, ou o valor `EOF`, em caso de falha.

Lendo caracteres

`int fgetc(FILE *arq)`

Obtém o próximo caractere da via apontada por `arq`, como um valor do tipo `unsigned char`; dessa forma, garante-se que todo caractere válido será positivo.

Valor de retorno. O código do caractere lido convertido em um valor do tipo `int`, se for bem sucedida, ou o valor `EOF`, em caso de falha.

`int getc(FILE *arq)`

Implementada como macro.

Lendo caracteres

Exemplo

```
#include <stdio.h>
int main(void) {
  FILE *arq;
  int c;
  arq = fopen("guararapes.txt", "r");
  while ((c = fgetc(arq)) != EOF) {
    printf("%d", c);
  }
  fclose(arq);
  return 0;
}
```

Retornando caracteres lidos

```
int ungetc(int c, FILE *arq)
```

Recoloca o caractere `c` na via de comunicação apontada por `arq`. O caractere é retornado à via de comunicação e o arquivo ao qual a via é associada permanece inalterado. O caractere é retornado como um valor do tipo `unsigned char` e será lido novamente pelo próximo comando de leitura.

Valor de retorno. O código do caractere retornado, convertido em um valor do tipo `int`, ou `EOF`, em caso de falha.

Retornando caracteres lidos

Exemplo

O programa ao lado lê e imprime caracteres, exceto os caracteres diferentes de ‘b’ digitados logo após um ‘a’ — estes não são impressos.

```
#include <stdio.h>
int main(void) {
  int c;
  while ((c = getc(stdin)) != '\n') {
    printf("obteve %c\n", c);
    if (c == 'a') {
      c = getc(stdin);
      if (c == 'b') {
        ungetc(c, stdin);
      }
    }
  }
  return 0;
}
```

Lendo linhas

```
char *fgets(char * restrict linha, int n,
                                        FILE * restrict arq)
```

Lê até `n` $-$ 1 caracteres da via de comunicação apontada `arq` , armazenando-os na cadeia de caracteres apontada por `linha`. A leitura é interrompida quando ocorre um fim de arquivo, quando o caractere de fim de linha é lido ou após a leitura de `n` $-$ 1 caracteres. O caractere de fim de linha que finaliza a leitura é armazenado na cadeia `linha`, bem como o caractere nulo, que é sempre colocado após o último caractere armazenado.

Valor de retorno. Um ponteiro para a cadeia `linha` ou o ponteiro nulo, se ocorrer um erro de leitura ou se ocorrer o fim de arquivo e nenhum caractere houver sido lido.

Lendo linhas

Exemplo

```
#include <stdio.h>
#define TAM (80)
int main(void) {
  FILE *arq;
  char linha[TAM];
  arq = fopen("alunos.txt", "r");
  while (fgets(linha, TAM, arq) != NULL) {
    printf("%s", linha);
  }
  fclose(arq);
  return 0;
}
```

Lendo linhas do teclado

A leitura de linhas do teclado deve especificar uma quantidade máxima de caracteres:

Usando scanf

```
scanf("%79[^\n]", linha)
```

Usando fgets

```
fgets(linha, TAM, stdin)
```

Lendo valores de tipos básicos

```
int fscanf(FILE * restrict arq,
                        const char * restrict formato, ...)
```

Lê do arquivo associado à via apontada por `arq` os valores que correspondem às diretivas de conversão presentes na cadeia apontada por `formato`, armazenando-os nas variáveis apontadas pelos argumentos da parte variável.

Valor de retorno. A quantidade de valores atribuídos ou `EOF`, em caso de falha.

Lendo valores de tipos básicos

Exemplo

```
#include <stdio.h>
int main(void) {
  char nome[31];
  int seq;
  double n1, n2, media;
  FILE *arq = fopen("alunos.txt", "r");
  nome[30] = '\0';
  while (fscanf(arq, "%d %30c%lf%lf\n",
               &seq, nome, &n1, &n2) != EOF)
  {
    media = (n1 + n2) / 2.0;
    if (media >= 7.0) {
      printf("%2d %30s %5.2f %5.2f %5.2f\n",
                  seq, nome, n1, n2, media);
    }
  }
  fclose(arq);
  return 0;
}
```

Gravando caracteres

`int fputc(int c, FILE *arq)`

Converte o caractere especificado por `c` em um valor do tipo `unsigned char` e o grava no arquivo associado à via apontada por `arq`.

Valor de retorno. O código do caractere gravado ou `EOF`, em caso de erro.

Gravando caracteres

`int fputc(int c, FILE *arq)`

Converte o caractere especificado por `c` em um valor do tipo `unsigned char` e o grava no arquivo associado à via apontada por `arq`.

Valor de retorno. O código do caractere gravado ou `EOF`, em caso de erro.

`int putc(int c, FILE *arq)`

Implementada como uma macro.

Gravando caracteres

Exemplo

```
#include <stdio.h>
#include <string.h>
#define QTD (30)
char *le_linha(char *, int);
int main(void) {
  FILE *arqE, *arqS;
  char nome[QTD], copia[QTD+6],
       sufixo[] = ".copia";
  int c;
  printf("Nome do arquivo: ");
  le_linha(nome, QTD);
  if ((arqE = fopen(nome,"r")) == NULL){
    printf("Arquivo %s inexiste\n", nome);
    return 0; /* fim programa */
  }
```

continua...

Gravando caracteres

Exemplo

...continuação

```
   for (int i = 0; i < strlen(nome); i++) {
     copia[i] = nome[i];
   }
   for (int i = 0; i < 7; i++) {
     copia[strlen(nome) + i] = sufixo[i];
   }
   if ((arqS = fopen(copia,"w")) == NULL){
     printf("Nao abriu copia\n");
     return 0; /* fim programa */
   }
   while ((c = fgetc(arqE)) != EOF) {
     fputc(c, arqS);
   }
   fclose(arqE); fclose(arqS);
   return 0;
 }
```

Gravando caracteres

Exemplo

...continuação

```
char *le_linha(char *linha, int n) {
  if (fgets(linha, n, stdin) != NULL) {
    if (linha[strlen(linha) - 1] == '\n') {
      linha[strlen(linha) - 1] = '\0';
    } else {
      scanf("%*[^\n]");
      scanf("%*c");
    }
    return linha;
  } else {
    return NULL;
  }
}
```

Gravando linhas

```
int fputs(const char * restrict linha, FILE * restrict arq)
```

Grava a cadeia de caracteres apontada por `linha` no arquivo associado à via apontada por `arq`. O caractere nulo que deve finalizar a cadeia não é gravado.

Valor de retorno. Um valor não-negativo ou `EOF`, em caso de erro.

Gravando linhas

Exemplo

```
#include <stdio.h>
#include <string.h>
char *altera(char *);
int main(void) {
  FILE *arq;
  arq = fopen("poema.trecho", "w");
  char tr1[] = "Atravessa esta paisagem";
  char tr2[8] = "oip!evn";
  fputs(tr1, arq);
  fputc(' ', arq);
  fputc('o', arq);
  fputs(" meu so", arq);
  fputs(altera(tr2), arq);
  fputs(" porto"" infinito", arq);
  fclose(arq);
  return 0;
}
```

continua...

Gravando linhas

Exemplo

...continuação

```
char *altera(char *str) {
  for (int i = 0; i < strlen(str); i++) {
    str[i]--;
  }
  return str;
}
```

Saída formatada

```
int fprintf(FILE * restrict arq,
                          const char * restrict formato, ...)
```

Grava no arquivo associado à via apontada por `arq` os valores armazenados nas variáveis apontadas pelos argumentos da parte variável, segundo as diretivas contidas na cadeia apontada por `formato`.

Valor de retorno. A quantidade de caracteres gravados ou um valor negativo, em caso de falha.

Saída formatada

Exemplo

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define QTD (31)
char *le_linha(char *, int);
void limpa_linha(void);
int main(void) {
  FILE *arq;
  char nome[QTD];
  double n1, n2;
  int seq = 0;
  arq = fopen("alunos.txt", "w");
```

continua...

Saída formatada

Exemplo

...continuação

```
  do {
    printf("Nome aluno ");
    printf("(<Enter> p/terminar): ");
    le_linha(nome, QTD);
    if (nome[0] == '\0') {
      break;
    }
    printf("Primeira nota: ");
    scanf("%lf", &n1);
    printf("Segunda nota: ");
    scanf("%lf", &n2);
    limpa_linha();
    fprintf(arq,"%2d %-30s %5.2f %5.2f\n",
                     ++seq, nome, n1, n2);
  } while (true);
  fclose(arq);
  return 0;
}
```

Gravando valores binários

```
size_t fwrite(const void * restrict vetor, size_t tam,
                              size_t qtd, FILE * restrict arq)
```

Grava no arquivo associado à via apontada por `arq` até `qtd` elementos do vetor apontado por `vetor`. Cada elemento é gravado com um tamanho igual a `tam` bytes.

Valor de retorno. A quantidade de elementos gravados. Se `qtd` ou `tam` for 0, o valor de retorno é 0 e o arquivo permanece inalterado. Nos demais casos o valor de retorno será diferente de `qtd` apenas se ocorrer algum erro de gravação.

Gravando valores binários

Exemplo

```
#include <stdio.h>
int main(void) {
  int nums[] = {12, 3, 23, 24, 6, 7};
  size_t qtd;
  FILE *arq = fopen("valores.bin", "ab");
  qtd = sizeof(nums)/sizeof(int);
  for (size_t i = 0; i < qtd; i++) {
    fwrite(nums + i, sizeof(int), 1, arq);
  }
  fclose(arq);
  return 0;
}
```

Gravando valores binários

Exemplo

```
#include <stdio.h>
int main(void) {
  int nums[] = {12, 3, 23, 24, 6, 7};
  size_t qtd;
  FILE *arq = fopen("valores.bin", "ab");
  qtd = sizeof(nums)/sizeof(int);
  for (size_t i = 0; i < qtd; i++) {
    fwrite(nums + i, sizeof(int), 1, arq);
  }
  fclose(arq);
  return 0;
}
```

A gravação pode ser substituída por:

```
fwrite(nums, sizeof(int), qtd, arq);
```

ou por

```
fwrite(nums, qtd * sizeof(int), 1, arq);
```

Gravando valores binários

Exemplo

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define QTD (31)
char *le_linha(char *, int);
void limpa_linha(void);
int main(void) {
  struct reg {
    int  seq;
    char nome[QTD];
    double n1, n2;
  } reg_aluno;
  FILE *arq;
  reg_aluno.seq = 0;
  arq = fopen("alunos.bin", "wb");
```

continua...

Gravando valores binários

Exemplo

...continuação

```
    do {
      printf("Nome aluno ");
      printf("(<Enter> p/terminar): ");
      le_linha(reg_aluno.nome, QTD);
      if (reg_aluno.nome[0] == '\0') {
        break;
      }
      (reg_aluno.seq)++;
      printf("Primeira nota: ");
      scanf("%lf", &(reg_aluno.n1));
      printf("Segunda nota: ");
      scanf("%lf", &(reg_aluno.n2));
      limpa_linha();
      fwrite(&reg_aluno,
               sizeof(struct reg), 1, arq);
    } while (true);
    fclose(arq);
    return 0;
  }
```

Lendo valores binários

```
size_t fread(const void * restrict vetor, size_t tam,
                          size_t qtd, FILE * restrict arq)
```

Lê do arquivo associado à via apontada por `arq` até `qtd` elementos de tamanho igual a `tam` bytes, armazenando-os no vetor apontado por `vetor`.

Valor de retorno. A quantidade de elementos lidos e armazenados no vetor. Se `tam` ou `qtd` for 0, a função retorna 0 e o conteúdo do vetor e o estado da via permanecem inalterados. Nos demais casos o valor de retorno é menor que `qtd` apenas se houver erro de leitura ou se o fim do arquivo for atingido.

Lendo valores binários

Exemplo

```
#include <stdio.h>
int main(void) {
  FILE *arq = fopen("valores.bin", "rb");
  int num[1];
  int qtd_num = 0; double media = 0.0;
  while (fread(num, sizeof(int),
                         1, arq) == 1) {
    media = media + num[0];
    qtd_num++;
  }
  media = media / qtd_num;
  printf("media = %f\n", media);
  fclose(arq);
  return 0;
}
```

Atualizando dados — modos de operação

`r+`, `rb+`, `r+b`.

- Abre um arquivo para atualização.
- O arquivo deve existir.
- O cursor pode ser reposicionado para leitura e gravação.

`w+`, `wb+`, `w+b`.

- Cria um arquivo vazio para atualização.
- Se o arquivo já existir, um novo arquivo vazio é criado, sobrepondo-se ao anterior.
- O cursor pode ser reposicionado para leitura e gravação.

`a+`, `ab+`, `a+b`.

- Cria ou abre um arquivo para atualização.
- Se o arquivo já existir, seus dados são preservados.
- O cursor pode ser reposicionado para leitura.
- A gravação ocorre sempre ao final do arquivo.

Definindo a posição do cursor

```
int fsetpos(FILE *arq, const fpos_t *pos)
```

Move o cursor de posição da via apontada por `arq` para a posição indicada pela estrutura apontada por `pos`.

Valor de retorno. Zero, se a operação é bem sucedida. Em caso de falha, a função retorna um valor diferente de zero e armazena um valor positivo em `errno`.

Obtendo a posição do cursor

```
int fgetpos(FILE *arq, const fpos_t *pos)
```

Armazena na estrutura apontada por `pos` as informações relativas à posição corrente da via apontada por `arq`.

Valor de retorno. Zero, se a operação é bem sucedida. Em caso de falha, a função retorna um valor diferente de zero e armazena um valor positivo em `errno`.

Usando fsetpos e fgetpos para atualizar arquivos

Esquema geral

1. Usar `fgetpos` para obter a posição do cursor antes de uma leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

Usando fsetpos e fgetpos para atualizar arquivos

Esquema geral

1. Usar `fgetpos` para obter a posição do cursor antes de uma leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   $\uparrow$

2. Realizar a leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

Usando fsetpos e fgetpos para atualizar arquivos

Esquema geral

1. Usar `fgetpos` para obter a posição do cursor antes de uma leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

2. Realizar a leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

3. Se o usuário optar por atualizar o registro, usar a função `fsetpos` para reposicionar o cursor na posição anterior

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

Usando fsetpos e fgetpos para atualizar arquivos

Esquema geral

1. Usar `fgetpos` para obter a posição do cursor antes de uma leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

2. Realizar a leitura

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

3. Se o usuário optar por atualizar o registro, usar a função `fsetpos` para reposicionar o cursor na posição anterior

   ...`6.32\n 2 Ines Pereira 8.3 6.15\n 3 Leonardo` ...

   ↑

4. Obter e gravar os novos dados

   ...`6.32\n 2 Ines Silva 6.2 7.21\n 3 Leonardo` ...

Usando fsetpos e fgetpos para atualizar arquivos

Exemplo

```
#include <stdio.h>
#include <stdbool.h>
#define QTD (31)
char escolhe_opcao(void);
void limpa_linha(void);
int main(void) {
  FILE *arq = fopen("alunos.txt","r+");
  fpos_t pos;
  char nome[31];
  double n1, n2;
  char opcao; int seq;
```

...continua

Usando fsetpos e fgetpos para atualizar arquivos

Exemplo

...continuação

```
while (true) {
  fgetpos(arq, &pos);
  if (fscanf(arq, "%d %30c %lf %lf%*c",
          &seq, nome, &n1, &n2) == EOF){
    break;
  }
  printf("%2d %30s %5.2f %5.2f\n",
                           seq, nome, n1, n2);
  opcao = escolhe_opcao();
  if (opcao == '9') {
    break;
  }
```

...continua

Usando fsetpos e fgetpos para atualizar arquivos

Exemplo

...continuação

```
switch (opcao) {
  case '1':
    printf("novo nome: ");
    scanf("%30[^\n]", nome);
    limpa_linha();
    printf("nova nota1: ");
    scanf("%lf", &n1);
    printf("nova nota2: ");
    scanf("%lf", &n2);
    limpa_linha();
    fsetpos(arq, &pos);
    fprintf(arq,
            "%2d %-30s %5.2f %5.2f\n",
                   seq, nome, n1, n2);
    break;
  default:
    printf("mantem os dados\n");
    break;
```

```
      } /* fim switch */
    }   /* fim while */
    fclose(arq);
    return 0;
}
```

...continua

Usando fsetpos e fgetpos para atualizar arquivos

Exemplo

...continuação

```
char escolhe_opcao(void) {
  char op;
  do {
    printf("Escolha a opcao\n");
    printf("1 - Atualiza\n");
    printf("2 - Mantem\n");
    printf("9 - Termina programa\n");
    printf("opcao: ");
    scanf("%c", &op);
    limpa_linha();
  } while ((op != '1') && (op != '2') &&
           (op != '9'));
  return op;
}
void limpa_linha(void) {
  scanf("%*[^\n]");
  scanf("%*c");
}
```

Reposicionando o cursor

```
int fseek(FILE *arq, long int deslocamento, int base)
```

Reposiciona o cursor da via apontada por `arq`, deslocando-o da quantidade de bytes indicada por `deslocamento` a partir da posição indicada por `base`. Os possíveis valores de `base` são:

- `SEEK_SET`. O cursor é reposicionado a partir do início do arquivo.
- `SEEK_CUR`. O cursor é reposicionado a partir da sua posição corrente.
- `SEEK_END`. O cursor é reposicionado a partir do fim do arquivo.

Valor de retorno. Zero, se a operação for bem sucedida, ou um valor diferente de zero, se o reposicionamento não puder ser realizado.

Obtendo a posição corrente e retornando ao início

```
long int ftell(FILE *arq)
```

Obtém a posição corrente do cursor da via apontada por `arq`.

Valor de retorno. Valor que corresponde à posição atual do cursor. Em caso de falha, a função retorna -1L e armazena um valor positivo em `errno`.

Obtendo a posição corrente e retornando ao início

```
long int ftell(FILE *arq)
```

Obtém a posição corrente do cursor da via apontada por `arq`.

Valor de retorno. Valor que corresponde à posição atual do cursor. Em caso de falha, a função retorna -1L e armazena um valor positivo em `errno`.

```
void rewind(FILE *arq)
```

Posiciona o cursor da via apontada por `arq` em seu início.

Valor de retorno. Não retorna valor.

Descarregando áreas de armazenamento temporário

```
int fflush(FILE *arq)
```

Descarrega a área de armazenamento temporário da via apontada por `arq`. Se a via é de gravação ou de atualização e a operação mais recente não é de entrada, então esta função força a gravação dos dados ainda não gravados. Nos demais casos o comportamento é indefinido.

Valor de retorno. Zero, se bem sucedida, ou `EOF`, em caso contrário.

Sincronizando arquivos e vias de comunicação

Quando um arquivo é aberto para atualização

- As operações de gravação não devem ser seguidas de uma operação de leitura,
  - sem que sua área de armazenamento temporário tenha sido descarregada (`fflush`), ou
  - sem que haja antes uma operação de posicionamento do cursor (`fseek`, `fsetpos` ou `rewind`).
- As operações de leitura não devem ser seguidas de uma operação de gravação,
  - sem que haja antes uma operação de posicionamento do cursor,
  - exceto se após a leitura o cursor apontar para o fim do arquivo.

Cadeias de caracteres como fonte e repositório

```
int sprintf(char * restrict cadeia,
                         const char * restrict formato, ...)
```

Grava na cadeia de caracteres apontada por `cadeia` os valores da parte variável dos argumentos, segundo as diretivas contidas na cadeia apontada por `formato`. O caractere nulo é inserido em `cadeia` imediatamente após a gravação do último argumento. O comportamento é indefinido se a gravação (incluindo o caractere nulo) extrapolar os limites da cadeia de caracteres.

Valor de retorno. A quantidade de caracteres gravados menos o caractere nulo inserido no fim da cadeia ou um valor negativo, se houver erro de formato.

Cadeias de caracteres como fonte e repositório

```
int snprintf(char * restrict cadeia, size_t n,
                         const char * restrict formato, ...)
```

Grava na cadeia de caracteres apontada por `cadeia` os valores da parte variável dos argumentos, segundo as diretivas contidas na cadeia apontada por `formato`, até o máximo de `n` − 1 caracteres. Todas as diretivas são avaliadas e todos os caracteres são produzidos, mas apenas os `n` − 1 caracteres iniciais são gravados, os demais são descartados. O caractere nulo é inserido imediatamente após a gravação do último argumento.

Valor de retorno. O número de caracteres da saída, antes da gravação. Isto é, o valor de retorno corresponde ao número de caracteres que seriam gravados se o valor de *n* fosse suficientemente grande. O valor de retorno é negativo caso ocorra algum erro de formato.

Cadeias de caracteres como fonte e repositório

```
int sscanf(char * restrict cadeia,
                        const char * restrict formato, ...)
```

Lê da cadeia de caracteres apontada por `cadeia` os valores que correspondem às diretivas da cadeia apontada por `formato`, armazenando-os nas variáveis indicadas na parte variável dos argumentos.

Valor de retorno. A quantidade de valores atribuídos ou `EOF`, caso haja algum erro antes de qualquer conversão.

Cadeias de caracteres como fonte e repositório

Exemplo

```
#include <stdio.h>
#include <float.h>
#define QTDC (35)
int main(void) {
  char formula[QTDC];
  double min = DBL_MAX, max = DBL_MIN;
  double num, soma = 0.0;
  int qtd = 0;
  printf("digite uma serie de numeros ");
  printf("(zero p/ terminar):\n");
  scanf("%lf", &num);
```

continua...

Cadeias de caracteres como fonte e repositório

Exemplo

...continuação

```
  while (num > 0.0) {
    qtd++;
    soma = soma + num;
    if (num < min) {
      min = num;
    }
    if (num > max) {
      max = num;
    }
    scanf("%lf", &num);
  }
  if (qtd > 0) {
    sprintf(formula, "%5.2f %5.2f %5.2f",
                    min, max, (soma / qtd));
    printf("cadeia= |%s|\n", formula);
  }
  return 0;
}
```

Usando funções de argumentos variáveis

As seguintes funções recebem seus argumentos de uma lista de argumentos variáveis, do tipo `va_list`:

Usando funções de argumentos variáveis

As seguintes funções recebem seus argumentos de uma lista de argumentos variáveis, do tipo `va_list`:

- `int vprintf(const char * restrict formato, va_list arg)`
  equivalente a `printf`.
- ```
  int vfprintf(FILE * restrict arq,
               const char * restrict formato, va_list arg)
  ```
  equivalente a `fprintf`.
  ```

Usando funções de argumentos variáveis

As seguintes funções recebem seus argumentos de uma lista de argumentos variáveis, do tipo `va_list`:

- `int vsprintf(char * restrict cadeia, const char * restrict formato, va_list arg)` equivalente a `sprintf`.
- `int vsnprintf(char * restrict cadeia, size_t n, const char * restrict formato, va_list arg)` equivalente a `snprintf`.

Usando funções de argumentos variáveis

As seguintes funções recebem seus argumentos de uma lista de argumentos variáveis, do tipo `va_list`:

- `int vscanf(const char * restrict formato, va_list arg)`
  equivalente a `scanf`.
- ```
  int vfscanf(FILE * restrict arq,
              const char * restrict formato, va_list arg)
  ```
  equivalente a `fscanf`.
- ```
  int vsscanf(const char * restrict cadeia,
              const char * restrict formato, va_list arg)
  ```
  equivalente a `ssscanf`.

Usando funções de argumentos variáveis

Nas funções que obtêm seus argumentos de uma lista de argumentos variáveis,

- a lista de argumentos deve ser iniciada com `va_start` e finalizada com `va_end`, e
- sempre que um argumento é consumido com a macro `va_arg` ele deixa de fazer parte da lista.

Usando funções de argumentos variáveis

Nas funções que obtêm seus argumentos de uma lista de argumentos variáveis,

- a lista de argumentos deve ser iniciada com va_start e finalizada com va_end, e
- sempre que um argumento é consumido com a macro va_arg ele deixa de fazer parte da lista.

Exemplo

```
void imp_vals(int qtd, ...) {
  va_list args;
  va_start(args, qtd);
  for (int i = 0; i < qtd; i++) {
    vprintf("%5.2f ", args);
    va_arg(args, double);
  }
  va_end(args);
}
```

Redirecionando as vias de comunicação

```
FILE *freopen(const char * restrict nome_arq,
            const char * restrict modo, FILE * restrict arq)
```

Abre o arquivo cujo nome é apontado por `nome_arq`, no modo apontado por `modo`, e o associa à via apontada por `arq`. A função tenta inicialmente fechar qualquer arquivo associado a `arq`, para só então proceder a abertura do novo arquivo no modo indicado. Se `nome_arq` é nulo, a função tenta modificar o modo de operação do arquivo associado à via apontada por `arq`.

Valor de retorno. O ponteiro `arq` ou o ponteiro nulo, se a operação falhar.

Redirecionando a entrada padrão

Exemplo

```
#include <stdio.h>
int main(void) {
  int nums[] = {2, 3, 10, 2, 14, 0, 11};
  FILE *arq;
  char nome_tmp[] = "teclado.txt";
  int val, soma = 0, qtd = 0;
  size_t lim = sizeof(nums)/sizeof(int);
  arq = fopen(nome_tmp, "w");
  for (size_t i = 0; i < lim; i++) {
    fprintf(arq, "%d ", nums[i]);
  }
  fclose(arq);
```

continua...

Redirecionando a entrada padrão

Exemplo

...continuação

```
  freopen(nome_tmp, "r", stdin);
  do {
    scanf("%d", &val);
    if (val > 0) {
      soma = soma + val;
      qtd++;
    }
  } while (val > 0);
  printf("media= %f\n",((double)soma)/qtd);
  return 0;
}
```

Usando arquivos e nomes temporários

```
FILE *tmpfile(void)
```

Cria um arquivo temporário binário, aberto para atualização no modo `wb+`. O arquivo é diferente de qualquer outro existente no ambiente de execução, sendo removido quando fechado ou ao término normal do programa.

Valor de retorno. Um ponteiro para a descrição do arquivo ou o ponteiro nulo, se o arquivo não pode ser criado.

Usando arquivos e nomes temporários

```
char *tmpnam(char *nome_arq)
```

Gera um nome que pode ser usado como nome de arquivo, pois será diferente de qualquer nome de arquivo existente no ambiente de execução.

Valor de retorno. Um ponteiro para a cadeia gerada, que aponta ou para a cadeia fornecida como argumento ou para uma variável estática interna. Se um nome não puder ser gerado, a função retorna o ponteiro nulo.

Exclusão e renomeação

```
int remove(const char *nome_arq)
```

Remove o arquivo de nome `nome_arq`. O arquivo deve estar fechado, o comportamento sobre um arquivo aberto é dependente da implementação.

Valor de retorno. Zero, se bem sucedida, ou um valor diferente de zero, em caso de falha.

```
int rename(const char *nome_antigo, const char *nome_novo)
```

Muda o nome do arquivo de nome `nome_antigo` para `nome_novo`. O arquivo deve estar fechado e não deve existir um arquivo com o novo nome, sendo o comportamento dependente da implementação, caso exista.

Valor de retorno. Zero, se bem sucedida, ou um valor diferente de zero, em caso de falha.

Leitura e gravação de caracteres multibytes

- O arquivo-cabeçalho `wchar.h` declara funções e macros que permitem a leitura e gravação de caracteres multibytes.
- O tipo `wchar_t` é usado para representar caracteres estendidos.
- O tipo `wint_t` é um tipo inteiro que pode representar todos os valores do tipo `wchar_t` e mais o valor `WEOF`.

Leitura e gravação de caracteres multibytes

Básica	Estendida/multibyte
getchar	wint_t getwchar(void)
fgetc	wint_t fgetwc(FILE *arq)
getc	wint_t getwc(FILE *arq)
ungetc	wint_t ungetwc(wint_t c, FILE *arq)
putchar	wint_t putwchar(wchar_t c)
fputc	wint_t fputwc(wchar_t c, FILE *arq)
putc	wint_t putwc(wchar_t c, FILE *arq)
fgets	wchar_t *fgetws(wchar_t * restrict linha, int n, FILE * restrict arq)
fputs	int fputws(const wchar_t * restrict linha, FILE * restrict arq)

Leitura e gravação de caracteres multibytes

Básica	Estendida/multibyte
printf	int wprintf(const wchar_t * restrict formato, ...)
scanf	int wscanf(const wchar_t * restrict formato, ...)
fprintf	int fwprintf(FILE * restrict arq, const wchar_t * restrict formato, ...)
fscanf	int fwscanf(FILE * restrict arq, const wchar_t * restrict formato, ...)
snprintf	int swprintf(wchar_t * restrict cadeia, size_t n, const wchar_t * restrict formato, ...)
sscanf	int swscanf(const wchar_t * restrict cadeia, const wchar_t * restrict formato, ...)

Leitura e gravação de caracteres multibytes

Básica	Estendida/multibyte
`vprintf`	`int vwprintf(const wchar_t * restrict formato, va_list arg)`
`vfprintf`	`int vfwprintf(FILE * restrict arq, const wchar_t * restrict formato, va_list arg)`
`vsnprintf`	`int vswprintf(wchar_t * restrict cadeia, const wchar_t * restrict formato, va_list arg)`
`vscanf`	`int vwscanf(const wchar_t * restrict formato, va_list arg)`
`vfscanf`	`int vfwscanf(FILE * restrict arq, const wchar_t * restrict formato, va_list arg)`
`vsscanf`	`int vswscanf(const wchar_t * restrict cadeia, const wchar_t * restrict formato, va_list arg)`

Leitura e gravação de caracteres multibytes

1. Definir a orientação da via de comunicação
   - para permitir que a leitura (ou gravação) de um caractere obtenha (ou produza) todos os bytes que o compõem.
2. Definir a localização utilizada na interpretação dos caracteres multibytes,
   - para permitir a correta conversão dos caracteres multibytes em estendidos, e vice-versa.
3. Utilizar as funções de leitura e gravação apropriadas.

Definindo a orientação das vias de comunicação

- Se a via não estiver orientada, sua primeira operação de leitura ou gravação determina a orientação.
- Uma vez definida, a orientação de uma via não pode ser modificada.
- A função `fwide` pode ser usada para determinar ou obter a orientação de uma via.

Definindo a orientação das vias de comunicação

```
int fwide(FILE *arq, int modo)
```

Determina ou obtém a orientação da via apontada por `arq`. Se `modo` for positivo a via será orientada a caracteres multibytes; se for negativo, a via será orientada a bytes; e se for igual a 0, a orientação não é modificada. Essa função não muda a orientação de uma via que já possua orientação.

Valor de retorno. Um valor positivo, se a via é orientada a caracteres multibytes, um valor negativo se a via é orientada a bytes, ou o valor 0, se a via não possui orientação. O valor de retorno corresponde à orientação da via após a execução da função, que pode ser idêntica à orientação original.

Definindo a orientação das vias de comunicação

Exemplo

O seguintes programas ilustram os vários modos de determinar a orientação de uma via.

Definindo a orientação das vias de comunicação

Exemplo

O seguintes programas ilustram os vários modos de determinar a orientação de uma via.

Orientação a bytes

```
#include <stdio.h>
#include <wchar.h>
int main(void) {
  char linha[80];
  printf("teclado: %d, video: %d\n",
      fwide(stdin, 0), fwide(stdout, 0));
  fgets(linha, 80, stdin);
  printf("teclado: %d, video: %d\n",
      fwide(stdin, 0), fwide(stdout, 0));
  return 0;
}
```

Definindo a orientação das vias de comunicação

Exemplo

O seguintes programas ilustram os vários modos de determinar a orientação de uma via.

Orientação a caracteres multibytes

```
#include <stdio.h>
#include <wchar.h>
int main(void) {
  wchar_t linha[80];
  wprintf(L"teclado: %d, video: %d\n",
      fwide(stdin, 0), fwide(stdout, 0));
  fgetws(linha, 80, stdin);
  wprintf(L"teclado: %d, video: %d\n",
      fwide(stdin, 0), fwide(stdout, 0));
  return 0;
}
```

Definindo a orientação das vias de comunicação

Exemplo

O seguintes programas ilustram os vários modos de determinar a orientação de uma via.

Orientação a bytes e a caracteres multibytes

```
#include <stdio.h>
#include <wchar.h>
int main(void) {
  wchar_t linha[80];
  printf("teclado: %d, video: %d\n",
    fwide(stdin, 20), fwide(stdout, -5));
  fgetws(linha, 80, stdin);
  printf("teclado: %d, video: %d\n",
    fwide(stdin, -2), fwide(stdout, 32));
  return 0;
}
```

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Identificação e tratamento de erros

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Erros de execução

Erros lógicos. Decorrentes de falhas na concepção.

Erros operacionais. Decorrentes do mau uso do programa.

Erros computacionais. Decorrentes de operações inválidas ou mau uso das funções.

Erros em funções da biblioteca padrão

Notificados de dois modos, que podem ocorrer separadamente ou em conjunto:

1. Através do valor de retorno das funções, indicando a ocorrência do erro.
2. Através da variável `errno` e de indicadores específicos para erros de entrada e saída.
    - O valor de `errno` é 0 no início da execução do programa,
    - Nenhuma função da biblioteca padrão zera essa variável antes de sua execução.
    - As funções que não usam `errno` para indicar erros podem alterar o seu conteúdo à vontade.

Erros em funções da biblioteca padrão

As seguintes macros, definidas no cabeçalho `errno.h`, correspondem a códigos de erro específicos que podem ser armazenados em `errno`:

`EDOM`. Indica erro nos argumentos das funções.

`ERANGE`. Indica erro no valor de retorno das funções.

`EILSEQ`. Indica erro na codificação de caracteres estendidos ou multibytes.

Erros de entrada e saída

```
void clearerr(FILE *arq)
```

Restaura o indicador de fim de arquivo e os demais indicadores de erro associados à via apontada por `arq` ao seu estado original (sem indicação de erro).

Valor de retorno. Não tem.

Erros de entrada e saída

```
void clearerr(FILE *arq)
```

Restaura o indicador de fim de arquivo e os demais indicadores de erro associados à via apontada por `arq` ao seu estado original (sem indicação de erro).

Valor de retorno. Não tem.

```
int ferror(FILE *arq)
```

Verifica o indicador de erro associado à via apontada por `arq`.

Valor de retorno. Um valor diferente de zero, se há algum erro indicado para a via, ou zero, em caso contrário.

Identificando o fim de arquivo

```
int feof(FILE *arq);
```

Verifica o indicador de fim de arquivo associado à via apontada por `arq`.

Valor de retorno. Um valor diferente de zero, se o indicador está ligado, ou zero, em caso contrário.

Identificando o fim de arquivo

As funções ferror e feof devem ser usadas para diferenciar um erro de entrada e saída da condição de fim de arquivo.

Exemplo

```
#include <stdio.h>
#include <stdlib.h>
#define QTD (31)
int main(void) {
  struct reg {
    int seq;
    char nome[QTD];
    double n1, n2;
  } aluno;
  size_t tam = sizeof(struct reg);
  FILE *arq = fopen("alunos.bin", "r");
  if (arq == NULL) {
    printf("Arq alunos.bin inexiste\n");
    return EXIT_FAILURE;
  }
```

continua...

Identificando o fim de arquivo

Exemplo

...continuação

```
   while (fread(&aluno, tam, 1, arq) == 1) {
     printf("%2d %-30s %5.2f %5.2f\n",
                   aluno.seq, aluno.nome,
                   aluno.n1, aluno.n2);
   }
   if (ferror(arq) != 0) {
     printf("Erro E/S: %d\n", ferror(arq));
     return EXIT_FAILURE;
   } else {
     fclose(arq);
     return EXIT_SUCCESS;
   }
}
```

Erros matemáticos

Erros de domínio. O valor de algum argumento está fora do domínio matemático da função.

Erros de imagem. O resultado da função não pode ser representado no tipo especificado para o valor de retorno, exceto com grande erro de arredondamento:

- Estouro por excesso. Quando a magnitude do resultado matemático é finita mas tão grande que não pode ser representada no tipo especificado.
- Estouro por falta. Quando a magnitude do resultado é tão pequena que o resultado não pode ser representado no tipo especificado.

Erros matemáticos

As seguintes funções ilustram a ocorrência de erros matemáticos:

Erros matemáticos

As seguintes funções ilustram a ocorrência de erros matemáticos:

```
double asin(double x)
```

$x \mapsto arcsen(x)$
$D = [-1, +1], Im = [-\pi/2, +\pi/2]$

`asin(2.0)` Resultado = `NAN`
Erro de domínio.

Erros matemáticos

As seguintes funções ilustram a ocorrência de erros matemáticos:

```
double tgamma(double x)
```

$$x \mapsto \Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$
$$D = \mathbb{R} - \{0, -1, -2, \ldots\}, Im = \mathbb{R}$$

`tgamma(DBL_MAX)`	Resultado = `HUGE_VAL` Erro de imagem (estouro por excesso).
`tgamma(INFINITY)`	Resultado = `HUGE_VAL` Não ocorre erro.

Erros matemáticos

As seguintes funções ilustram a ocorrência de erros matemáticos:

```
double exp2(double x)
```

$x \mapsto 2^x$

$D = \mathbb{R}, Im = (0, +\infty]$

`exp2(-DBL_MAX)`	Resultado $= 0$ (por conta do arredondamento) Erro de imagem (estouro por falta).
`exp2(-INFINITY)`	Resultado $= 0$ Sem erros

Notificação de erros matemáticos

As funções matemáticas de ponto flutuante notificam os erros dos seguintes modos, que podem ocorrer separadamente ou em conjunto:

1. Através da variável `errno`:

`EDOM`	Erros de domínio.
`ERANGE`	Erros de imagem.

2. Através das exceções de ponto flutuante definidas pelas seguintes macros no cabeçalho `fenv.h`:

`FE_INVALID`	Valores inválidos.
`FE_DIVBYZERO`	Divisão por zero ou estouro por excesso (em situações especiais).
`FE_OVERFLOW`	Estouro por excesso.
`FE_UNDERFLOW`	Estouro por falta.
`FE_INEXACT`	Erros de arredondamento.

Testando o modo de notificação em vigor

1. Se `math_errhandling & MATH_ERRNO` é diferente de zero:
   - `errno` = `EDOM` para erros de domínio
   - Para erros de imagem decorrentes de:
   - `errno` = `ERANGE` para erros de imagem decorrentes de estouro por excesso.
   - Para erros de imagem decorrentes de estouro por falta, a atribuição `errno` = `ERANGE` pode não ocorrer.
2. Se `math_errhandling & MATH_ERREXCEPT` é diferente de zero:
   - Para erros de domínio, a exceção `FE_INVALID` é ativada.
   - Para erros de imagem decorrentes de:
     - Estouro por excesso. A exceção `FE_DIVBYZERO` é ativada, se o resultado é um valor infinito obtido por aproximação com argumentos finitos. `FE_OVERFLOW` é ativada, nos demais casos.
     - Estouro por falta. A ativação da exceção `FE_UNDERFLOW` é dependente da implementação, pode não ocorrer.

Restaurando e inspecionando as exceções

```
int feclearexcept(int tipos_excecao)
```

Restaura as exceções indicadas no argumento, fazendo com que elas fiquem desativadas. O argumento tanto pode ser uma macro indicando uma exceção particular, como `FE_INVALID`, quanto a disjunção de várias macros indicando o conjunto delas, como `FE_INVALID | FE_INEXACT`. A macro `FE_ALL_EXCEPT` provê a disjunção de todas as exceções de ponto flutuante definidas para o ambiente.

Valor de retorno. Zero, se todas as exceções indicadas no argumento forem restauradas (ou se o argumento for zero), ou um valor diferente de zero, em caso de falha.

Restaurando e inspecionando as exceções

```
int fetestexcept(int tipos_excecao)
```

Verifica se a exceção indicada no argumento está ativa. Se o argumento for uma disjunção de várias exceções, a função verifica se cada uma está ativa.

Valor de retorno. Retorna a disjunção das configurações de cada exceção ativa indicada no argumento. O valor retornado é uma configuração de bits armazenada em um valor do tipo `int` é será zero apenas se nenhuma das exceções indicadas estiver ativa.

Restaurando e inspecionando as exceções

As macros de exceção de ponto flutuante são implementadas como potências de 2. Desse modo podem ser definidas e inspecionadas através dos operadores de conjunção e disjunção binárias.

Restaurando e inspecionando as exceções

Exemplo. Para as seguintes configurações:

Macro	Valor	Configuração
FE_INVALID	1	000001
FE_DIVBYZERO	2	000010
FE_OVERFLOW	4	000100
FE_UNDERFLOW	8	001000
FE_INEXACT	16	010000

Se `resultado = FE_INVALID | FE_UNDERFLOW | FE_INEXACT` pode-se usar a conjunção binária para verificar se uma determina configuração está contida nessa variável:

```
resultado & FE_INVALID                        = 000001
resultado & FE_INEXACT                        = 010000
resultado & (FE_INVALID | FE_UNDERFLOW)       = 001001
resultado & (FE_OVERFLOW | FE_UNDERFLOW)      = 001000
resultado & FE_DIVBYZERO                      = 000000
```

Restaurando e inspecionando as exceções

Exemplo

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>
void imp_erros(void);
int main(void) {
  double x;
  while (scanf("%lf", &x) != EOF) {
    errno = 0;
    feclearexcept(FE_ALL_EXCEPT);
    printf("exp(%g)  = %g ", x, exp(x));
    imp_erros();
    errno = 0;
    feclearexcept(FE_ALL_EXCEPT);
    printf("gama(%g) = %g ", x, tgamma(x));
    imp_erros();
  }
  return 0;
}
```

continua

Restaurando e inspecionando as exceções

Exemplo

...continuação

```
void imp_erros(void) {
  if (MATH_ERRNO) {
    if (errno == EDOM) {
      printf ("  Nao definida");
    }
  }
  if (MATH_ERREXCEPT) {
    if (fetestexcept(FE_UNDERFLOW) == FE_UNDERFLOW) {
      printf ("  Estouro por falta");
    }
    if (fetestexcept(FE_OVERFLOW) & FE_OVERFLOW) {
      printf ("  Estouro por excesso");
    }
  }
  printf ("\n");
}
```

Erros de precisão, conversão e arredondamento

As operações em ponto flutuante podem produzir resultados imprecisos devido

- à imprecisão na representação binária dos valores e
- ao limite na quantidade de bits usada para armazená-los.

Erros de precisão, conversão e arredondamento

Exemplo

Para uma arquitetura de 32 bits e valores do tipo `double`, o código gerado pelo compilador gcc torna a expressão

```
(3.0 / 30.0) == ((1.0 / 3.0) * (6.0 / 20.0))
```

falsa, enquanto que

```
(3.0 / 30.00) == ((1.0 * 6.0) / (3.0 * 20.0))
```

é verdadeira. A diferença é devida a simplificações e à ordem de avaliação, que produz resultados intermediários diferentes.

Erros de precisão

A macro DBL_EPSILON costuma ser usada para implementar em C o teste de igualdade entre valores do tipo double.

Erro absoluto

```
if (fabs(x - y) <= DBL_EPSILON) {
  /* x igual a y */
}
```

Erro relativo

```
if (fabs(x - y) <= (fabs(y) * DBL_EPSILON)) {
  /* x igual a y */
}
```

Erros de conversão

- As conversões entre tipos diferentes podem produzir valores errados.
- Nas conversões de tipos reais de ponto flutuante em tipos inteiros,
  - se o valor real é infinito ou `NAN`, ou se sua parte inteira excede os limites do tipo alvo,
    - uma exceção de ponto flutuante inválida (`FE_INVALID`) é ativada, sendo o resultado da conversão não especificado.
  - Se o valor real não é inteiro, mesmo que sua parte inteira possa ser representada no tipo alvo, pode ocorrer uma exceção de ponto flutuante inexata (`FE_INEXACT`), dependendo da implementação.

Erros de conversão

Exemplo

O programa ao lado produz a seguinte saída:

```
De:  inf Para:  -2147483648
-> valor invalido
De:  235.016693 Para:  235
-> valor inexato
```

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>
void imp_erros(void);
int main(void) {
  int numi;
  double numf = 3.0/0.0;
  feclearexcept(FE_ALL_EXCEPT);
  numi = numf;
  printf("De: %f Para: %d ", numf, numi);
  imp_erros();
  numf = 235.0167;
  feclearexcept(FE_ALL_EXCEPT);
  numi = numf;
  printf("De: %f Para: %d ", numf, numi);
  imp_erros();
  return 0;
}
```

continua...

Erros de conversão

Exemplo. *...continuação*

```
void imp_erros(void) {
  int excecao;
  if (!MATH_ERREXCEPT) {
    return;
  }
  excecao = fetestexcept(FE_OVERFLOW | FE_UNDERFLOW | FE_INVALID |
                         FE_INEXACT | FE_DIVBYZERO);
  if (excecao == 0) {
    printf(" -> Sem excecao\n");
    return;
  }
  if (excecao & FE_UNDERFLOW) {
     printf(" -> estouro por falta");
  if (excecao & FE_OVERFLOW) {
     printf(" -> estouro por excesso");
  if (excecao & FE_INVALID) {
     printf(" -> valor inexato");
  if (excecao & FE_DVIBYZERO) {
     printf(" -> estouro/ vlr infinito");
  printf("\n");
}
```

Arredondamento

As seguintes macros, declaradas no cabeçalho `fenv.h`, definem os possíveis modos de arredondamento:

`FE_TONEAREST` Arredonda para o valor mais próximo.

Contrário ao zero. Arredonda para o valor mais próximo. Se o número estiver exatamente no meio entre dois valores, será escolhido o mais distante do zero. Embora não exista uma macro específica para esse modo, ele é adotado pelas funções `round`, `lround` e `llround`.

`FE_TOWARDZERO` Arredonda em direção ao zero. Se o número for positivo será arredondado para menos, e se for negativo será arredondado para mais.

`FE_UPWARD` Arredonda em direção ao infinito positivo. O arredondamento será sempre para mais.

`FE_DOWNWARD` Arredonda em direção ao infinito negativo. O arredondamento será sempre para menos.

Arredondamento

	Modo de arredondamento				
número	mais próximo	para zero	para $+\infty$	para $-\infty$	contrário ao zero
$2,1$	2	2	3	2	2
$2,5$	2	2	3	2	3
$3,5$	4	3	4	3	4
$3,9$	4	3	4	3	4
$-2,1$	-2	-2	-2	-3	-2
$-2,5$	-2	-2	-2	-3	-3
$-3,5$	-4	-3	-3	-4	-4
$-3,9$	-4	-3	-3	-4	-4

Arredondamento

```
int fegetround(void)
```

Inspeciona o modo de arredondamento.

Valor de retorno. Um valor não-negativo, referente à macro que define o modo de arredondamento em vigor, ou um valor negativo, se o modo de arredondamento não puder ser determinado.

```
int fesetround(int modo)
```

Modifica o modo de arredondamento para o modo indicado por `modo`. Se o argumento não corresponde a um modo válido, nenhuma mudança é efetuada.

Valor de retorno. Zero, se o modo foi modificado, ou um valor diferente de zero, em caso contrário.

Sinais de interrupção

SIGTERM *Término*. O programa é interrompido por solicitação de término.

SIGABRT *Término anormal*. O programa é interrompido por falhas que impedem o prosseguimento da execução.

SIGINT *Interrupção/Interação*. O programa é interrompido por solicitação resultante da interação com o ambiente de execução.

SIGFPE *Exceção de ponto flutuante*. O programa é interrompido ao tentar realizar uma operação matemática inválida.

SIGILL *Instrução ilegal*. O programa é interrompido ao tentar executar uma instrução ilegal, provavelmente por corrupção da área onde reside o código do programa.

SIGSEGV *Violação de segmento*. O programa é interrompido ao tentar acessar uma área indevida da memória.

Sinais de interrupção

Síncrono. Quando o sinal é gerado durante a operação (ou evento) que o origina, antes da execução da próxima operação (ou da ocorrência do próximo evento).

- A maioria dos sinais decorrentes de erros computacionais é síncrona.
- Em algumas implementações o sinal `SIGFPE` pode ser assíncrono.
- Os sinais decorrentes de `raise` e `abort` são síncronos.

Assíncrono. Quando o sinal pode ser gerado algumas operações (ou eventos) após a operação (ou evento) que o origina.

- Sinais decorrentes de outros processos ou eventos externos são assíncronos.

Sinais de interrupção

Quando uma operação ou evento gera um sinal de interrupção ele fica pendente até ser enviado ao programa que, ao recebê-lo, executa uma das seguintes ações:

- Ignora o sinal
- Executa uma ação padrão
- Executa a função registrada para o tratamento do sinal

Registrando funções de tratamento de sinais

```
void (*signal(int sinal, void (*funcao)(int)))(int)
```

Registra a função apontada por `funcao` para tratar o sinal indicado por `sinal`.

Valor de retorno. O ponteiro para a função de tratamento que estava anteriormente associada ao sinal, se a nova função for corretamente registrada, ou `SIG_ERR`, em caso de falha.

A função de tratamento de sinal deve ser do tipo `void (int)`.

Registrando funções de tratamento de sinais

O tratamento de sinais possui muitos aspectos dependentes da implementação:

- Quando uma função para tratamento de sinal é chamada, o tratamento padrão para o sinal que originou a chamada é restabelecido.
  - Entretanto, algumas implementações podem apenas bloquear o envio dos sinais de mesmo tipo do que está sendo tratado, enquanto durar a função de tratamento.
- Após a execução de uma função de tratamento de sinal, o controle pode ou não voltar ao ponto seguinte ao ponto de chamada, reiniciando o processamento a partir do próximo comando ou operação.

Registrando funções de tratamento de sinais

Exemplo

```
#include <stdio.h>
#include <signal.h>
void trata_erro(int);
int main(void) {
  int a = -6137;
  signal(SIGABRT, trata_erro);
  signal(SIGTERM, trata_erro);
  signal(SIGINT, trata_erro);
  signal(SIGILL, trata_erro);
  signal(SIGSEGV, trata_erro);
  signal(SIGFPE, trata_erro);
  do {
    printf("Envie sinal p/este programa.");
    scanf("%d", &a);
    printf("Valor de a = %d\n", a);
  } while (a < 0);
  return 0;
}
```

continua...

Registrando funções de tratamento de sinais

Exemplo

...continuação

```
void trata_erro(int sinal) {
  signal(sinal, trata_erro);
  printf("Recebeu sinal %d. ", sinal);
}
```

Lançando sinais de interrupção

```
int raise(int sig)
```

Lança o sinal de erro correspondente ao seu argumento.

Valor de retorno. Zero, se bem sucedida, ou um valor diferente de zero, em caso de falha.

Lançando sinais de interrupção

`int raise(int sig)`

Lança o sinal de erro correspondente ao seu argumento.

Valor de retorno. Zero, se bem sucedida, ou um valor diferente de zero, em caso de falha.

`_Noreturn void abort(void)`

Lança o sinal `SIGABRT` que provoca o término anormal do programa.

Valor de retorno. Não tem.

Tratando interrupções

As seguintes restrições são estabelecidas pelo padrão da linguagem:

- Uma função de tratamento de sinal não deve chamar `raise` se o sinal que a ativou foi em decorrência de uma execução de `abort` ou `raise`.
- Uma função de tratamento de sinal não deve chamar funções da biblioteca padrão, exceto as funções `_Exit`, `abort`, `raise` e `signal`.
- A função `signal` pode ser chamada no corpo de uma função de tratamento de sinal apenas para o sinal que originou sua execução.
- As variáveis estáticas referidas por uma função de tratamento de sinal devem ser do tipo `volatile sig_atomic_t`.

Tratando interrupções

Encerrando o processamento

- O padrão da linguagem estabelece que a execução deve ser finalizada para os sinais `SIGFPE`, `SIGILL` e `SIGSEGV` (*comportamento é indefinido, caso o controle retorne*).
- Se uma função desenvolvida para encerrar o processamento de modo controlado, for registrada para vários sinais, pode-se usar uma variável estática do tipo `volatile sig_atomic_t` para evitar a interrupção do tratamento.

Tratando interrupções

Encerrando o processamento

Exemplo

```
#include <stdio.h>
#include <stdbool.h>
#include <signal.h>
void trata_fpe(int);
FILE *arq;
int main(void) {
  int num, den;
  arq = fopen("arqtst", "w");
  if (signal(SIGFPE, trata_fpe) ==
                                SIG_ERR) {
    printf("sem tratamento erro: FPE\n");
  }
```

continua...

Tratando interrupções

Encerrando o processamento

Exemplo

...continuação

```
    while (true) {
      printf("Digite numerador");
      printf(" (0 p/terminar): ");
      scanf("%d", &num);
      if (num == 0) {
        break;
      }
      printf("Digite denominador: ");
      scanf("%d", &den);
      fprintf(arq, "%d %d %d\n",
                          num, den, num/den);
    }
    fclose(arq);
    return 0;
  }
```

continua...

Tratando interrupções

Encerrando o processamento

Exemplo

```
volatile sig_atomic_t encerrando = false;

void trata_fpe(int s) {
  if (encerrando) {
    raise(s);
  }
  encerrando = true;
  printf("Erro sinal %d\n", s);
  fclose(arq);
  signal(s, SIG_DFL);
  raise(s);
}
```

Tratando interrupções

Encerrando o processamento

Exemplo

```
volatile sig_atomic_t encerrando = false;

void trata_fpe(int s) {
  if (encerrando) {
    raise(s);
  }
  encerrando = true;
  printf("Erro sinal %d\n", s);
  fclose(arq);
  signal(s, SIG_DFL);
  raise(s);
}
```

OBS. Esta função de tratamento de sinal não está de acordo com o padrão, pois faz uso das funções `printf` e `fclose`.

Tratando interrupções

Retomando o processamento

- Quando o processamento é retomado após o tratamento de um sinal, ele é reiniciado a partir do ponto em que ocorreu a interrupção (*e não do ponto em que a operação ou o evento que gerou o sinal ocorreu*).
- Se for necessário diferenciar o fluxo normal do fluxo após a ocorrência do erro, deve-se usar variáveis estáticas do tipo `volatile sig_atomic_t`.

Tratando interrupções

Retomando o processamento

Exemplo

```
#include <stdio.h>
#include <stdbool.h>
#include <signal.h>
void trata_int(int);
volatile sig_atomic_t erro;
int main(void) {
  int num, soma = 0;
  signal(SIGINT, trata_int);
  erro = false;
  do {
    scanf("%d", &num);
    if ((num == 0) || (erro)) {
      break;
    }
    soma = soma + num;
    /* codigo omitido */
  } while (true);
  printf("soma = %d\n", soma);
  return 0;
}
void trata_int(int s) {
  signal(s, trata_int);
  erro = true;
}
```

Usando desvios não locais

Esquema geral

- A macro `setjmp`, geralmente em uma estrutura de decisão, marca o ponto ao qual o fluxo de execução deve retornar.
  - O valor de retorno igual a 0 indica um processamento sem erros.
  - O valor de retorno diferente de zero indica uma condição de erro.
- Durante o processamento, sempre que uma condição de erro for detectada, a função `longjmp` é chamada, tendo como segundo argumento o código que provoca o desvio para a rotina de tratamento de erro apropriada.

Desvios não locais e funções de tratamento de sinais

- O padrão da linguagem especifica que o comportamento é indefinido se `longjmp` é chamada no interior de uma função de tratamento de sinais.
  - Entretanto, muitas implementações permitem esse tipo de chamada, pois ele era válido no padrão anterior (ISO/IEC 9899:1990).
- O desvio não local é usado (nas implementações que o permitem) para tratar sinais que normalmente causam o término da execução.

Desvios não locais e funções de tratamento de sinais

Exemplo

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
jmp_buf estado;
void erro_divzero(int);
```

```
int main(void) {
  int a, b, res = 0;
  signal(SIGFPE, erro_divzero);
  while (setjmp(estado) >= 0) {
    printf("Digite a: ");
    scanf("%d", &a);
    printf("Digite b: ");
    scanf("%d", &b);
    res = a / b;
    break;
  }
  printf("resultado a/b = %d\n", res);
  return 0;
}
void erro_divzero(int sinal) {
  printf("Erro: div por zero.\n");
  printf("Entre com novos valores.\n");
  signal(SIGFPE, erro_divzero);
  longjmp(estado, 1);
}
```

Assertivas

`void assert(`⟨*expressão escalar*⟩`)`

Avalia a expressão fornecida como argumento. Se a expressão é verdadeira a execução prossegue normalmente; se a expressão é falsa imprime uma mensagem de erro e interrompe a execução com uma chamada a `abort`.

Valor de retorno. Não tem.

Assertivas

`void assert(`⟨*expressão escalar*⟩`)`

Avalia a expressão fornecida como argumento. Se a expressão é verdadeira a execução prossegue normalmente; se a expressão é falsa imprime uma mensagem de erro e interrompe a execução com uma chamada a `abort`.

Valor de retorno. Não tem.

- A interrupção provocada pela macro `assert` não permite o tratamento da condição de erro, exceto se uma função para tratar o sinal `SIGABRT` for registrada.
- A definição da macro `NDEBUG` torna as assertivas inativas.

Assertivas

Exemplo

No trecho de programa ao lado a função `analisaAeB` é interrompida sempre que a expressão `(b > 0) || (a < 3 * b)` for falsa.

```
void analisaAeB(int a, int b) {
  assert((b > 0) || (a < 3 * b));
  /* codigo omitido */
}
```

Descrevendo os erros

```
char *strerror(int cod_erro)
```

Seleciona a mensagem que corresponde ao erro cujo código é igual a `cod_erro`.

Valor de retorno. Um ponteiro para a cadeia que contém a mensagem selecionada.

Descrevendo os erros

`char *strerror(int cod_erro)`

Seleciona a mensagem que corresponde ao erro cujo código é igual a `cod_erro`.

Valor de retorno. Um ponteiro para a cadeia que contém a mensagem selecionada.

`void perror(const char *prefixo)`

Grava na saída padrão de erro (`stderr`) a cadeia de caracteres indicativa do erro cujo código está armazenado na variável `errno`, antecedida da cadeia `prefixo`, se esta for não nula.

Valor de retorno. Não tem.

Recomendações

Recomendação 1 Declarar vetores com tamanho fixo, usando macros para definir o tamanho.
Razão. Favorece a consistência do programa.

Recomendação 2 Evitar o uso de funções que não permitem determinar o tamanho da cadeia. Deve-se preferir `fgets` a `gets`, por exemplo.
Razão. Evita o acesso indevido à memória.

Recomendação 3 Criar cadeias delimitadas com o caractere nulo.
Razão. As cadeias que não possuem o caractere nulo ao final não podem ser usadas nas funções da biblioteca padrão que assumem a existência desse caractere.

Recomendações

Recomendação 4 Evitar conversão de valores.
Razão. As conversões indevidas podem modificar os valores originais.

Recomendação 5 Validar os valores obtidos com funções de leitura.
Razão. As funções de leitura nem sempre produzem valores válidos, podendo falhar ou resultar em conversões indevidas.

Recomendação 6 Declarar variáveis com valor inicial.
Razão. As variáveis automáticas não iniciadas, podem assumir qualquer valor quando o bloco que as contém é ativado pela primeira vez.

Recomendações

Recomendação 7 Assegurar que os valores são válidos antes do uso de um ponteiro.
Razão. Um ponteiro com valor nulo ou indevido remete a um endereço inválido.

Recomendação 8 Verificar a compatibilidade de tipos das declarações que referem-se ao mesmo objeto.
Razão. As declarações que referem-se a um mesmo objeto devem ter tipos compatíveis.

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Caracteres e cadeias de caracteres

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Tamanho

```
size_t strlen(const char *cadeia)
```

Calcula o tamanho da cadeia apontada por `cadeia`, que deve ser terminada pelo caractere nulo (não incluído no cálculo).

Valor de retorno. O tamanho da cadeia.

Tamanho

`size_t strlen(const char *cadeia)`

Calcula o tamanho da cadeia apontada por `cadeia`, que deve ser terminada pelo caractere nulo (não incluído no cálculo).

Valor de retorno. O tamanho da cadeia.

Exemplo

```
int conta_letra(char c, char *cda) {
  int qtd = 0;
  for (size_t i = 0; i < strlen(cda); i++) {
    if (cda[i] == c) {
      qtd++;
    }
  }
  return qtd;
}
```

Cópia

```
char *strcpy(char * restrict dest,
                                 const char * restrict orig)
```

Copia a cadeia apontada por `orig`, incluindo o caractere nulo que a finaliza, para a cadeia apontada por `dest`. As cadeias não podem estar sobrepostas, e a cadeia de origem deve ser terminada pelo caractere nulo.

Valor de retorno. O ponteiro `dest`.

Cópia

```
char *strncpy(char * restrict dest,
                         const char * restrict orig, size_t qtd)
```

Copia até `qtd` caracteres da cadeia apontada por `orig` para a cadeia apontada por `dest`. Se a cadeia de origem é menor que `qtd` caracteres, caracteres nulos são inseridos em `dest` até completar a quantidade especificada; se é maior ou igual, apenas os `qtd` caracteres iniciais da origem são copiados. As cadeias não podem estar sobrepostas e a cadeia de origem deve terminar com o caractere nulo, se for menor que `qtd` caracteres.

Valor de retorno. O ponteiro `dest`.

Cópia

Exemplo

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char linha[16];
  char copia[20] = "@@@@@@@@@@@@@@@@@@@";
  size_t qtd;
  scanf("%15[^\n]", linha);
  scanf("%*[^\n]"); scanf("%*c");
  printf("Qtd caracteres p/copia: ");
  scanf("%zu", &qtd);
  strncpy(copia, linha, qtd);
  printf("Origem : |%s|\n", linha);
  printf("Destino: |");
```

continua...

Cópia

Exemplo

...continuação

```
  for (int i = 0; i < 20; i++) {
    if (copia[i] == '\0') {
      putchar('^');
    } else {
      putchar(copia[i]);
    }
  }
  putchar('|');
  return 0;
}
```

Concatenação

```
char *strcat(char * restrict cesq,
                                    const char * restrict cdir)
```

Concatena as cadeias `cesq` e `cdir`, anexando uma cópia da cadeia apontada por `cdir` ao fim da cadeia apontada por `cesq`. As duas cadeias devem ser terminadas pelo caractere nulo, e não devem se sobrepor. Na cadeia resultante a cópia do caractere inicial de `cdir` sobrepõe-se ao caractere nulo de `cesq`.

Valor de retorno. O ponteiro `cesq`.

Concatenação

```
char *strncat(char * restrict cesq,
                     const char * restrict cdir, size_t qtd)
```

Concatena as cadeias `cesq` e `cdir`, anexando uma cópia dos `qtd` caracteres iniciais da cadeia apontada por `cdir` ao fim da cadeia apontada por `cesq`. A concatenação termina após a cópia de `qtd` caracteres ou quando o caractere nulo de `cdir` é encontrado. As cadeias não devem se sobrepor e a cadeia `cesq` deve ser terminada pelo caractere nulo, assim como a cadeia `cdir`, se seu tamanho for menor do que `qtd` caracteres. Na cadeia resultante a cópia do caractere inicial de `cdir` sobrepõe-se ao caractere nulo de `cesq`. Um caractere nulo é sempre inserido no fim da cadeia resultante.

Valor de retorno. O ponteiro `cesq`.

Concatenação

Exemplo

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void) {
  char prefixo[21] = "\0";
  char sufixo[21] = "\0";
  char nome[41] = "\0";
  printf("prefixo: ");
  scanf("%20[^\n]", prefixo);
  scanf("%*[^\n]"); scanf("%*c");
  printf("sufixo : ");
  scanf("%20[^\n]", sufixo);
  strcpy(nome, prefixo);
  strcat(nome, sufixo);
  printf("nome   : |%s|\n", nome);
  return 0;
}
```

Comparação

```
int strcmp(const char *cesq, const char *cdir)
```

Compara a cadeia de caracteres apontada por `cesq` com a cadeia apontada por `cdir`. As duas cadeias devem ser terminadas com o caractere nulo. Os caracteres são comparados como valores do tipo `unsigned char`. O primeiro caractere que for maior que o caractere correspondente da outra cadeia, faz com que sua cadeia seja maior. Se as cadeias tiverem tamanhos diferentes, com todos os caracteres iguais até o menor dos tamanhos, então a cadeia de maior tamanho é maior.

Valor de retorno. Zero, se as cadeias forem iguais; um valor negativo se `cesq` for menor do que `cdir`; ou um valor positivo, se `cesq` for maior do que `cdir`.

Comparação

```
int strncmp(const char *cesq, const char *cdir, size_t qtd)
```

Compara os caracteres iniciais das cadeias apontadas por `cesq` e `cdir`, até o máximo de `qtd` caracteres. Para os `qtd` caracteres iniciais (ou para todos os caracteres, se uma cadeia possui menos caracteres) a comparação é realizada do modo descrito na função `strcmp`.

Valor de retorno. Zero, se as cadeias forem iguais; um valor negativo se `cesq` for menor do que `cdir`; ou um valor positivo, se `cesq` for maior do que `cdir`.

Comparação

Exemplo

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
int main(void) {
  char menor[20] = "\0";
  char palavra[20] = "\0";
  printf("Digite algumas palavras\n");
  printf("(Ctrl-d para terminar):\n");
  scanf("%19s", menor);
  do {
    if (scanf("%19s", palavra) == EOF) {
      break;
    }
    if (strcmp(palavra, menor) < 0) {
      strcpy(menor, palavra);
    }
  } while (true);
  printf("Menor palavra = %s\n", menor);
  return 0;
}
```

Pesquisa

```
char *strchr(const char *cadeia, int c)
```

Procura a primeira ocorrência do caractere `c`, convertido em um valor do tipo `char`, na cadeia apontada por `cadeia`, que deve ser terminada pelo caractere nulo.

Valor de retorno. Ponteiro para a primeira ocorrência do caractere na cadeia ou o ponteiro nulo, se o caractere não estiver presente.

Pesquisa

```
char *strrchr(const char *cadeia, int c)
```

Procura a última ocorrência do caractere `c`, convertido em um valor do tipo `char`, na cadeia apontada por `cadeia`, que deve ser terminada pelo caractere nulo.

Valor de retorno. Ponteiro para a última ocorrência do caractere na cadeia ou o ponteiro nulo, se o caractere não estiver presente.

Pesquisa

Exemplo

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char verso[] = "Na verde rama do amor!";
  char ini;
  char *esq, *dir;
  printf("Digite uma letra: ");
  scanf("%c", &ini);
  esq = strchr(verso, ini);
  dir = strrchr(verso, ini);
  if (esq != NULL) {
    printf("Maior: %s\n", esq);
  }
  if (dir != NULL) {
    printf("Menor: %s\n", dir);
  }
  return 0;
}
```

Pesquisa

```
char *strstr(const char *cesq, const char *cdir)
```

Procura na cadeia apontada por `cesq` a primeira ocorrência da cadeia apontada por `cdir`. As cadeias devem ser terminadas com o caractere nulo, mas o caractere nulo de `cdir` não é incluído na busca.

Valor de retorno. O ponteiro para o início da primeira ocorrência de `cdir` em `cesq` ou o ponteiro nulo, se `cdir` não está contida em `cesq`. Se o comprimento de `cdir` é zero, o valor de retorno é `cesq`.

Pesquisa

```
size_t strspn(const char *cesq, const char *cdir)
```

Calcula o comprimento do maior segmento inicial da cadeia apontada por `cesq` consistindo apenas de caracteres que constam da cadeia apontada por `cdir`. As cadeias devem ser terminadas com o caractere nulo.

Valor de retorno. O comprimento do segmento.

Pesquisa

```
size_t strcspn(const char *cesq, const char *cdir)
```

Calcula o comprimento do maior segmento inicial da cadeia apontada por `cesq` consistindo apenas de caracteres que não constam da cadeia apontada por `cdir`. As cadeias devem ser terminadas com o caractere nulo.

Valor de retorno. O comprimento do segmento.

Pesquisa

Exemplo

A função subcadeia retorna o índice da primeira ocorrência da cadeia sub em txt, ou um valor maior que o tamanho de txt, se sub não estiver contida em txt.

```
ptrdiff_t subcadeia(char txt[], char sub[]) {
  char *res = strstr(txt, sub);
  if (res == NULL) {
    return (ptrdiff_t)(strlen(txt) + 1);
  } else {
    return (ptrdiff_t)(res - txt);
  }
}
```

Decomposição

```
char *strtok(char * restrict cadeia,
                                const char * restrict delim)
```

Obtém os formantes da cadeia apontada por `cadeia`, usando como delimitadores os caracteres da cadeia apontada por `delim`. Quando a função é chamada com `cadeia` diferente de nulo, o primeiro formante é obtido. Uma chamada subsequente com `cadeia` igual a nulo obtém o próximo formante da mesma cadeia usada na chamada anterior.

Valor de retorno. Ponteiro para o formante obtido na chamada ou o ponteiro nulo, se não houver formantes.

Decomposição

```
char *strtok(char * restrict cadeia,
                                  const char * restrict delim)
```

Obtém os formantes da cadeia apontada por `cadeia`, usando como delimitadores os caracteres da cadeia apontada por `delim`. Quando a função é chamada com `cadeia` diferente de nulo, o primeiro formante é obtido. Uma chamada subsequente com `cadeia` igual a nulo obtém o próximo formante da mesma cadeia usada na chamada anterior.

Valor de retorno. Ponteiro para o formante obtido na chamada ou o ponteiro nulo, se não houver formantes.

OBS. Esta função modifica o conteúdo de `cadeia`.

Decomposição

Exemplo

O programa ao lado obtém os formantes constituintes de uma data no formato

$\langle dia \rangle / \langle mes \rangle / \langle ano \rangle$

```
#include <stdio.h>
#include <string.h>
_Bool valida_data(int, char *,
                       char *, char *);
int main(void) {
  char linha[31] = "\0";
  char *dia, *mes, *ano;
  int erro_delim = 0;
  scanf("%30[^\n]", linha);
  if ((strspn(linha, "/") > 0) ||
      (strstr(linha, "//") != NULL)) {
      erro_delim = 1;
  }
  dia = strtok(linha, "/");
  mes = strtok(NULL, "/");
  ano = strtok(NULL, "");
  valida_data(erro_delim, dia, mes, ano);
  return 0;
}
```

Conversões reais

```
double strtod(const char * restrict num,
                                    char ** restrict resto)
```

Converte a cadeia de caracteres apontada por `num` em um valor do tipo `double`. Um ponteiro para a cadeia restante, composta pelos caracteres finais a partir do primeiro não utilizado, é armazenado na variável apontada por `resto`, se `resto` não é nulo.

Valor de retorno. Os seguintes valores são retornados:

(a) O valor do tipo `double` que corresponde à cadeia convertida, se a conversão ocorre sem erros.

(b) O valor `HUGE_VAL` sinalizado, se ocorre um estouro por excesso. Nesse caso, o valor da macro `ERANGE` é atribuído a `errno`.

(c) Um valor dependente da implementação, se ocorre um estouro por falta. Nesse caso, a atribuição de `ERANGE` a `errno` depende da implementação.

(d) O valor 0 se a conversão não é possível. Nesse caso, o valor do ponteiro `num` é armazenado na variável apontada por `resto`, se `resto` não é nulo.

Conversões reais

```
float strtof(const char * restrict num,
                                char ** restrict resto)
```

```
long double strtold(const char * restrict num,
                                char ** restrict resto)
```

As funções `strtof` e `strtold` são equivalentes a `strtod`.

Conversões reais

```
float strtof(const char * restrict num,
                                     char ** restrict resto)
```

```
long double strtold(const char * restrict num,
                                     char ** restrict resto)
```

As funções `strtof` e `strtold` são equivalentes a `strtod`.

```
double atof(const char *num)
```

A função `atof` corresponde a `strtod(num, (char **)NULL)`. Entretanto, ela não precisa indicar a ocorrência de erros em `errno` e seu comportamento é indefinido, se o resultado não puder ser representado como um valor do tipo `double`.

Conversões reais

Exemplo

O trecho de código ao lado produz os seguintes resultados:

```
char cadeia[30]; char *resto;
errno = 0;
double num = strtod(cadeia, &resto);
```

cadeia	num	resto	errno
`"   12"`	12	`""`	0
`"   13.536"`	$13,536$	`""`	0
`"   13.536 a 12 "`	$13,536$	`" a 12 "`	0
`"5e-23 "`	5×10^{-23}	`" "`	0
`"a5e2"`	0	`"a5e2"`	0
`"4.2p2"`	$4,2$	`"p2"`	0
`"0x4.2p2"`	$16,5 = (4 + 2 \times 16^{-1}) \times 2^2$	`""`	0
`"1e400"`	inf	`""`	ERANGE
`"1e-315 "`	1×10^{-315}	`" "`	ERANGE

Conversões inteiras

```
long int strtol( const char * restrict num,
                          char ** restrict resto, int base)
```

Converte a cadeia apontada por `num` em um valor do tipo `long int`. Um ponteiro para a cadeia restante, composta pelos caracteres finais a partir do primeiro não utilizado, é armazenado na variável apontada por `resto`, se `resto` não é nulo.

Valor de retorno. Os seguintes valores são retornados:

(a) O valor do tipo `long int` que corresponde à cadeia convertida, se a conversão ocorre sem erros.

(b) O valor `LONG_MIN` ou `LONG_MAX`, se o valor convertido não pode ser representado no tipo especificado. Nesse caso o valor da macro `ERANGE` é atribuído a `errno`.

(c) O valor 0 se a conversão não é possível. Nesse caso, o valor do ponteiro `num` é armazenado na variável apontada por `resto`, se `resto` não é nulo.

Conversões inteiras

```
long long int strtoll(const char * restrict num,
                              char ** restrict resto, int base)
```

```
unsigned long int strtoul(const char * restrict num,
                              char ** restrict resto, int base)
```

```
unsigned long long int strtoull(const char * restrict num,
                              char ** restrict resto, int base)
```

As funções `strtoll`, `strtoul` e `strtoull` são equivalentes à função `strtol`.

Conversões inteiras

```
int atoi(const char *num)
```

Corresponde a `(int)strtol(num, (char **)NULL, 10)`.

```
long int atol(const char *num)
```

Corresponde a `strtol(num, (char **)NULL, 10)`.

```
long long int atoll(const char *num)
```

Corresponde a `strtoll(num, (char **)NULL, 10)`.

As funções `atoi`, `atol` e `atoll` não precisam indicar a ocorrência de erros. O comportamento é indefinido, caso o resultado não possa ser representado como um valor do tipo especificado por elas.

Conversões inteiras

Exemplo

O trecho de código ao lado produz os seguintes resultados:

```
char cadeia[30]; char *resto;
int base;
errno = 0;
double num = strtol(cadeia, &resto, base);
```

cadeia	base	num	resto	errno
`"  101231  "`	10	101.231	`"  "`	0
`"  101231  "`	2	$5 = 1 \times 2^2 + 1 \times 2^0$	`"231  "`	0
`"034a bc"`	10	34	`"a bc"`	0
`"034a bc"`	8	$28 = 3 \times 8^1 + 4 \times 8^0$	`"a bc"`	0
`"025"`	0	$21 = 2 \times 8^1 + 5 \times 8^0$	`""`	0
`"025"`	8	$21 = 2 \times 8^1 + 5 \times 8^0$	`""`	0
`"025"`	10	25	`""`	0
`"0x25"`	0	$37 = 2 \times 16^1 + 5 \times 16^0$	`""`	0
`"0x25"`	16	$37 = 2 \times 16^1 + 5 \times 16^0$	`""`	0
`"25"`	16	$37 = 2 \times 16^1 + 5 \times 16^0$	`""`	0
`"25"`	0	25	`""`	0
`"2147483648"`	10	2.147.483.647	`""`	ERANGE

Classificação de caracteres

As funções de classificação de caracteres têm a forma geral `int fun(int c)`, retornando verdadeiro, se o caractere pertencer à categoria indicada pela função, e falso, em caso contrário.

Função	Categoria	Função	Categoria
`isblank`	(*Branco*)	`isalpha`	(*Alfabético*)
`isspace`	(*Espaço*)	`isalnum`	(*Alfanumérico*)
`isdigit`	(*Dígito*)	`ispunct`	(*Pontuação*)
`isxdigit`	(*Dígito hex*)	`isgraph`	(*Gráfico*)
`islower`	(*Minúsculo*)	`isprint`	(*Imprimível*)
`isupper`	(*Maiúsculo*)	`iscntrl`	(*Controle*)

Mapeamento de caracteres

```
int tolower(int c)
```

Valor de retorno. Um caractere minúsculo correspondente, se existir. Em caso contrário, o caractere `c` é retornado sem modificação.

```
int toupper(int c)
```

Valor de retorno. Um caractere maiúsculo correspondente, se existir. Em caso contrário, o caractere `c` é retornado sem modificação.

Caracteres estendidos e multibytes

- Os cabeçalhos `wchar.h` e `wctype.h` declaram funções para o tratamento de caracteres e cadeias de caracteres multibytes.
- Para a maioria da funções que lidam com caracteres do conjunto básico de caracteres existe uma função correspondente que lida com caracteres multibytes.

Caracteres estendidos e multibytes

- Os cabeçalhos `wchar.h` e `wctype.h` declaram funções para o tratamento de caracteres e cadeias de caracteres multibytes.
- Para a maioria da funções que lidam com caracteres do conjunto básico de caracteres existe uma função correspondente que lida com caracteres multibytes.

Exemplo

Básica	Multibyte	Básica	Multibyte
`strcpy`	`wcscpy`	`strncpy`	`wcsncpy`
`strcmp`	`wcscmp`	`strncmp`	`wcsncmp`
`strchr`	`wcschr`	`strtok`	`wcstok`
`strtod`	`wcstod`	`strtol`	`wcstol`
`isblank`	`iswblank`	`isdigit`	`iswdigit`
`tolower`	`towlower`	`toupper`	`towupper`

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Utilitários e funções matemáticas

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Alocação e realocação de espaços de memória

```
void *malloc(size_t tam)
```

Aloca espaço de memória de tamanho igual a `tam` bytes. O conteúdo do espaço alocado é indeterminado.

Valor de retorno. Ponteiro para o endereço inicial do espaço alocado ou o ponteiro nulo, em caso de falha.

Alocação e realocação de espaços de memória

```
void *malloc(size_t tam)
```

Aloca espaço de memória de tamanho igual a `tam` bytes. O conteúdo do espaço alocado é indeterminado.

Valor de retorno. Ponteiro para o endereço inicial do espaço alocado ou o ponteiro nulo, em caso de falha.

```
void *calloc(size_t qtd, size_t tam)
```

Aloca espaço de memória suficiente para armazenar `qtd` elementos de `tam` bytes cada. Todos os bits do espaço alocado são iniciados com zeros.

Valor de retorno. Ponteiro para o endereço inicial do espaço alocado ou o ponteiro nulo, em caso de falha.

Alocação e realocação de espaços de memória

```
void *realloc(void *ptr, size_t tam)
```

Desaloca o espaço apontado por `ptr`, realocando seu conteúdo em um novo espaço de tamanho igual a `tam` bytes. Se

- `tam` é maior que o espaço apontado por `ptr`, o conteúdo dos bytes excedentes é indeterminado;
- `tam` é menor, apenas os `tam` bytes iniciais são copiados.

O comportamento é indeterminado se `ptr` não aponta para um espaço previamente alocado por `malloc`, `calloc` ou `realloc` (ou se aponta para um espaço que tenha sido desalocado com as funções `free` ou `realloc`).

Valor de retorno. Ponteiro para o endereço inicial do novo espaço alocado ou o ponteiro nulo, em caso de falha.

Alocação e realocação de espaços de memória

Exemplo

O programa ao lado aloca espaços em blocos de ($10 \times$ tamanho do tipo `int`), sempre que necessário.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int *numeros = NULL;
  int num, tam = 0, qtd = 0;
  printf("Digite os numeros ");
  printf("(zero p/terminar):\n");
  scanf("%d", &num);
  while (num != 0) {
    qtd++;
    if (qtd > tam) {
      tam = tam + 10;
      numeros = (int *)realloc(numeros,
                       tam * sizeof(int));
    }
    numeros[qtd - 1] = num;
    scanf("%d", &num);
  }
```

OBS. a chamada `realloc(NULL, tam)` é equivalente a `malloc(tam)`.

continua...

Alocação e realocação de espaços de memória

Exemplo

...continuação

```
    printf("Numeros armazenados:\n");
    for (int i = 0; i < qtd; i++) {
      printf("%3d ", numeros[i]);
    }
    return 0;
}
```

Liberação de espaços de memória

- O vazamento de memória é caracterizado pela existência de espaço de memória alocado, mas que não pode ser acessado.
- Para evitar os vazamentos de memória, todo espaço alocado pelas funções `malloc`, `calloc` e `realloc`, que não seja mais necessário, deve ser explicitamente desalocado.

Liberação de espaços de memória

- O vazamento de memória é caracterizado pela existência de espaço de memória alocado, mas que não pode ser acessado.
- Para evitar os vazamentos de memória, todo espaço alocado pelas funções `malloc`, `calloc` e `realloc`, que não seja mais necessário, deve ser explicitamente desalocado.

```
void free(void *ptr)
```

Desaloca o espaço apontado pelo ponteiro `ptr`, que deve apontar para um espaço previamente alocado por `malloc`, `calloc` ou `realloc`; caso contrário, o comportamento é indefinido.

Valor de retorno. Não tem.

Cópia e movimentação de espaços de memória

```
void *memcpy(void * restrict dest,
                        const void * restrict orig, size_t qtd)
```

Copia `qtd` bytes do espaço de memória apontado por `orig` para o espaço de memória apontado por `dest`. O espaço destino deve comportar os bytes copiados. Se houver sobreposição dos espaços, o comportamento é indefinido.

Valor de retorno. O ponteiro `dest`.

Cópia e movimentação de espaços de memória

```
void *memmove(void *dest, const void *orig, size_t qtd)
```

Copia `qtd` bytes do espaço de memória apontado por `orig` para o espaço de memória apontado por `dest`. O espaço destino deve comportar os bytes copiados. Os espaços de origem e destino podem se sobrepor.

Valor de retorno. O ponteiro `dest`.

Cópia e movimentação de espaços de memória

Exemplo

O programa ao lado sobrepõe elementos do vetor num, a partir do seu início.

```
#include <stdio.h>
#include <string.h>
int main(void) {
  int num[5] = {10, 5, 9, 4, 7};
  int ind;
  do {
    printf("Digite 0 < ind < 5: ");
    scanf("%d", &ind);
  } while ((ind < 1) || (ind > 4));
  printf("original    : ");
  for (int i = 0; i < 5; i++) {
    printf("%d, ", num[i]);
  }
  memmove((void *)num, (void *)(num+ind),
              (5 - ind) * sizeof(int));
  printf("\nmodificado : ");
  for (int i = 0; i < 5; i++) {
    printf("%d, ", num[i]);
  }
  return 0;
}
```

Comparação de espaços de memória

```
int memcmp(const void *esq, const void *dir, size_t qtd)
```

Compara os `qtd` bytes iniciais dos espaços de memória apontados por `esq` e `dir`. Os bytes são comparados como valores do tipo `unsigned char`.

Valor de retorno. Zero, se os espaços forem iguais, um valor negativo se `esq` for menor que `dir`, ou um valor positivo se `esq` for maior que `dir`.

Pesquisa de espaços de memória

```
void *memchr(const void *obj, int c, size_t qtd)
```

Procura a primeira ocorrência do caractere `c`, convertido em um valor do tipo `unsigned char`, nos `qtd` bytes iniciais do espaço apontado por `obj`. Os bytes do espaço de memória pesquisado são interpretados como valores do tipo `unsigned char`.

Valor de retorno. Ponteiro com o endereço do caractere ou o ponteiro nulo, se o caractere não estiver no espaço pesquisado.

Modificação de espaços de memória

```
void *memset(void *mem, int c, size_t qtd)
```

Copia o caractere `c`, convertido em um valor do tipo `unsigned char`, para cada um dos `qtd` bytes iniciais do espaço apontado por `mem`.

Valor de retorno. O ponteiro `mem`.

Ordenamento de vetores

```
void qsort(void *vetor, size_t qtd, size_t tam,
                   int (*fcomp)(const void *, const void *))
```

Ordena o vetor de `qtd` elementos de tamanho `tam`, apontado por `vetor`, usando a ordem ascendente determinada pela função de comparação `fcomp`.

Valor de retorno. Não tem.

Ordenamento de vetores

A função de comparação usada na função `qsort` deve receber dois argumentos do tipo ponteiro para `void` e retornar

- um valor negativo, se o conteúdo apontado pelo primeiro argumento for menor que o conteúdo apontado pelo segundo;
- um valor positivo, se for maior;
- ou o valor zero, se forem iguais.

Os elementos do vetor não podem ser modificados pela função de comparação, e para dois elementos iguais não é especificado qual deles virá à esquerda no vetor ordenado.

Ordenamento de vetores

Exemplo

O programa ao lado ordena os primeiros qtd elementos de um vetor de até 1.000 elementos.

```
#include <stdio.h>
#include <stdlib.h>
#define TAM (1000)
int compara(const void *, const void *);
int main(void) {
  int nums[TAM];
  int qtd = 0;
  printf("Digite ate 1000 nums inteiros");
  printf(" (Ctrl-d para terminar) \n");
  while ((qtd < TAM) &&
          scanf("%d", &(nums[qtd]))!=
          EOF)
  {qtd++;}
  qsort(nums, qtd, sizeof(int), compara);
  for (int i = 0; i < qtd; i++) {
    printf("%d ", nums[i]);
  }
  return 0;
}
```

continua...

Ordenamento de vetores

Exemplo

...continuação

```
int compara(const void *numa,
            const void *numb) {
  if (*(int *)numa < *(int *)numb) {
    return -1;
  } else {
    if (*(int *)numa > *(int *)numb) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Pesquisa em vetores

```
void *bsearch(const void *chv, const void *vetor,
                                   size_t qtd, size_t tam,
                   int (*fcomp)(const void *, const void *))
```

Pesquisa o vetor de `qtd` elementos de tamanho `tam`, apontado por `vetor`, verificando se algum de seus elementos possui a chave apontada por `chv`. A função de comparação usada para comparar a chave de cada elemento com aquela apontada por `chv` é fornecida pelo ponteiro `fcomp`.

Valor de retorno. Um ponteiro para o primeiro elemento do vetor que possui a chave igual à especificada ou o ponteiro nulo, se nenhum possuir chave igual à especificada. Se mais de um elemento possuir chave igual à especificada, qualquer um pode ser apontado pelo valor de retorno.

Pesquisa em vetores

A função de comparação usada na função `bsearch` deve receber dois argumentos do tipo ponteiro para `void` e retornar

- um valor negativo, se o conteúdo do primeiro argumento for menor que o conteúdo do segundo;
- um valor positivo, se for maior; ou
- o valor zero, se forem iguais.

Os elementos do vetor devem estar ordenados e não podem ser modificados pela função de comparação.

Pesquisa em vetores

Exemplo

O programa ao lado estende o exemplo anterior. Após ordenar os números, ele pesquisa no vetor ordenado os números informados pelo usuário.

```
#include <stdio.h>
#include <stdlib.h>
#define TAM (1000)
int compara(const void *, const void *);
int main(void) {
  int nums[TAM];
  int qtd = 0, val;
  int *ptr_val;
  printf("Digite ate 1000 nums inteiros");
  printf(" (Ctrl-d para terminar) \n");
  while ((qtd < TAM) &&
          scanf("%d", &(nums[qtd]))!=
          EOF)
  {qtd++;}
  qsort(nums, qtd, sizeof(int), compara);
  for (int i = 0; i < qtd; i++) {
    printf("%d ", nums[i]);
  }
```

continua...

Pesquisa em vetores

Exemplo

...continuação

```
  printf("\nDigite numeros p/pesquisa:\n");
  while (scanf("%d", &val) != EOF) {
    ptr_val = bsearch((void *)&val, nums,
                      qtd, sizeof(int), compara);
    if (ptr_val != NULL) {
      printf("%d esta no vetor\n", val);
    } else {
      printf("%d fora do vetor\n", val);
    }
  }
  return 0;
}
```

continua...

Pesquisa em vetores

Exemplo

...continuação

```
int compara(const void *numa,
            const void *numb) {
  if (*(int *)numa < *(int *)numb) {
    return -1;
  } else {
    if (*(int *)numa > *(int *)numb) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Localização

Localização de uma fonte de dados é a adaptação das informações dessa fonte às convenções adotadas por determinado idioma ou país.

- Localização do ambiente de execução.
- Localização adotada por um programa.
  - No início da execução a localização padrão C é adotada.
  - Pode ser modificada com a função `setlocale`.
- Localização da via de comunicação.
  - Corresponde à localização do programa no momento em que a orientação da via é definida.
  - A localização dos dados corresponde à localização da via que foi utilizada em sua gravação.

Categorias de localização

LC_ALL	Refere-se a todos os aspectos de uma localização.
LC_COLLATE	Aspectos relacionados ao ordenamento dos caracteres.
LC_CTYPE	Aspectos relacionados à codificação dos caracteres.
LC_MONETARY	Aspectos monetários da informação.
LC_NUMERIC	Aspectos numéricos (não monetários) da informação.
LC_TIME	Aspectos relacionados ao tempo.

Definindo a localização

```
char *setlocale(int categoria, const char *local)
```

Se a cadeia apontada por `local` é nula, a função apenas identifica a localização em vigor no programa para a categoria especificada, sem modificá-la. Se a cadeia apontada por `local` não é nula, a função atribui aos aspectos da localização especificados por `categoria` as convenções da localização identificada por `local`.

Valor de retorno. Depende da cadeia que identifica a localização:

- `local` não é nula. Retorna a cadeia de caracteres que identifica a nova localização para a categoria especificada, se a nova localização foi estabelecida com sucesso, ou o ponteiro nulo, em caso de falha.
- `local` é nula. Retorna a cadeia de caracteres que identifica a localização em vigor associada à categoria especificada.

Definindo a localização

- O valor "C" identifica a localização padrão.

```
setlocale(LC_ALL, "C");
```

- A cadeia vazia identifica a localização nativa do ambiente de execução.

```
setlocale(LC_ALL, "");
```

- A localização não modifica a representação interna dos valores, apenas o modo como sua representação textual é interpretada.

Definindo a localização

Exemplo

O programa ao lado lê e imprime dois valores usando diferentes localizações.

```
#include <stdio.h>
#include <locale.h>
int main(void) {
  double v1, v2;
  char *local;
  local = setlocale(LC_NUMERIC, NULL);
  printf("localizacao atual: %s\n", local);
  printf("Valor (use ponto decimal): ");
  scanf("%lf", &v1);
  local = setlocale(LC_NUMERIC, "pt_BR");
  printf("localizacao atual: %s\n", local);
  printf("Valor (use virgula decimal): ");
  scanf("%lf", &v2);
  printf("Valores: %g e %g\n", v1, v2);
  return 0;
}
```

Definindo a codificação utilizada

A localização da via determina como os caracteres multibytes são interpretados:

```c
#include <stdio.h>
#include <string.h>

int main(void) {
  char linha[80];

  fgets(linha, 80, stdin);
  printf("%s (tam=%zd)\n",
        linha, strlen(linha));
  for (size_t i = 0;
       i < strlen(linha);
       i++)
  {
    printf("%c|", linha[i]);
  }
  printf("fim\n");
  return 0;
}
```

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
int main(void) {
  wchar_t linha[80];
  setlocale(LC_ALL, "");
  fgetws (linha, 80, stdin);
  wprintf(L"%ls (tam=%zd)\n",
        linha, wcslen(linha));
  for (size_t i = 0;
       i < wcslen(linha);
       i++)
  {
    wprintf(L"%lc|", linha[i]);
  }
  wprintf(L"fim\n");
  return 0;
}
```

Inspecionando a localização

Os aspectos numéricos e monetários de uma localização podem ser inspecionados com a função:

```
struct lconv *localeconv(void)
```

Armazena nos componentes de uma estrutura do tipo `struct lconv` os símbolos usados na formatação dos valores numéricos e monetários.

Valor de retorno. Uma estrutura com as convenções de formatação numérica e monetária para a localização em vigor.

Inspecionando a localização

Alguns componentes da estrutura `struct lconv`:

`char *decimal_point`	Símbolo do separador decimal.
`char *thousands_sep`	Símbolo do separador de milhar.
`mon_decimal_point`	Símbolo do separador decimal.
`mon_thousands_sep`	Símbolo do separador de milhar.
`char frac_digits`	Quantidade de dígitos decimais (após o separador decimal).
`char *currency_symbol`	Símbolo indicador da moeda.
`char p_cs_precedes`	Posição do indicador da moeda para valor não-negativo: 1, se o indicador precede o valor, ou 0, se o sucede.

Inspecionando a localização

Exemplo

A função ao lado imprime o conteúdo de alguns componentes da localização em vigor.

```
void imp_locale() {
  struct lconv *L = localeconv();
  printf("ponto decimal: %s\n",
                              L->decimal_point);
  printf("separador milhar: %s\n",
                              L->thousands_sep);
  printf("ponto decimal monetario: %s\n",
                         L->mon_decimal_point);
  printf("separador milhar monetario:%s\n",
                         L->mon_thousands_sep);
  printf("simbolo monetario: %s\n",
                            L->currency_symbol);
}
```

Informações de tempo

Tempo local. São a data e hora de cada lugar.

Tempo médio. São a data e hora locais adotadas no meridiano 0.

- Tempo médio de Greenwich (GMT)
- Tempo universal (UT)
- Tempo solar médio

Tempo universal coordenado. São a data e hora calculadas com base em padrões atômicos, em vez do cálculo com base no movimento da terra.

- Também é referido pela sigla UTC (do inglês *Universal Time, Coordinated*). A hora zero UTC corresponde aproximadamente a meia-noite GMT.

Tempo de calendário. São a data e hora calculadas a partir de um marco inicial, definido politicamente.

- Tempo de sala é o tempo de calendário transcorrido entre dois eventos

Representações do tempo

- O tipo `time_t` é um tipo aritmético usado para representar o tempo de calendário (o marco adotado pelo compilador gcc é 0h0m0 de 01/01/1970).
- Os tempos universal e local são representados pelo tipo `struct tm`:

`int tm_sec`	segundos após o minuto	[0, 60]
`int tm_min`	minutos após a hora	[0, 59]
`int tm_hour`	horas desde a meia noite	[0, 23]
`int tm_mday`	dia do mês	[1, 31]
`int tm_mon`	meses desde janeiro	[0, 11]
`int tm_year`	anos desde 1900	
`int tm_wday`	dias desde domingo	[0, 6]
`int tm_yday`	dias desde 1 de janeiro	[0, 365]
`int tm_isdst`	indicador de horário de verão	

Obtendo data e hora

```
time_t time(time_t *tempo)
```

Determina o tempo de calendário corrente. O valor de retorno também é armazenado na variável apontada pelo parâmetro `tempo`, se este não é nulo.

Valor de retorno. tempo de calendário (como um valor `time_t`) ou -1, se o tempo de calendário não está disponível.

Obtendo data e hora

```
struct tm *gmtime(const time_t *tempo)
```

Converte o tempo de calendário apontado por `tempo` no tempo universal coordenado (UTC) correspondente, armazenando-o em uma estrutura do tipo `struct tm`.

Valor de retorno. Um ponteiro para a estrutura gerada ou o ponteiro nulo, se o tempo especificado não puder ser convertido.

Obtendo data e hora

```
struct tm *localtime(const time_t *tempo)
```

Converte o tempo de calendário apontado por `tempo` no tempo local correspondente, armazenando-o em uma estrutura do tipo `struct tm`. A conversão considera o fuso e o horário de verão definidos no ambiente de execução.

Valor de retorno. Um ponteiro para a estrutura gerada ou o ponteiro nulo, se o tempo especificado não puder ser convertido.

Obtendo data e hora

```
time_t mktime(struct tm *tempo)
```

Converte o valor da estrutura apontada por `tempo`, considerada como um tempo local, em um valor do tipo `time_t`.

Valor de retorno. O tempo local especificado, codificado como um valor do tipo `time_t`, ou -1, se o tempo especificado não puder ser convertido.

Obtendo data e hora

Exemplo

O programa ao lado converte o tempo corrente e o converte nos tempos local e universal.

```
#include <stdio.h>
#include <time.h>
void imp_tempo(struct tm *);
int main(void) {
  struct tm *t_local, *t_utc;
  time_t t_atual;
  t_atual = time(NULL);
  printf("tempo atual: %ld\n", t_atual);
  t_local = localtime(&t_atual);
  imp_tempo(t_local);
  t_utc = gmtime(&t_atual);
  imp_tempo(t_utc);
  return 0;
}
```

continua...

Obtendo data e hora

Exemplo

...continuação

```
void imp_tempo(struct tm *t) {
  char *sem[7] = {"dom", "seg", "ter",
                  "qua", "qui", "sex", "sab"};
  printf("%d h %d min %d s %d/%d/%d ",
              t->tm_hour, t->tm_min, t->tm_sec,
              t->tm_mday, (t->tm_mon + 1),
              (t->tm_year + 1900));
  if ((t->tm_wday < 0) || (t->tm_wday > 6))
    printf("(%d, ", t->tm_wday);
  else
    printf("(%s, ", sem[t->tm_wday]);
  printf("%d dias%s)\n", t->tm_yday,
          t->tm_isdst > 0 ? ", verao" : "");
}
```

Obtendo data e hora

Exemplo

O programa ao lado lê os valores de um dia, mês e ano e mostra o valor do tipo time_t correspondente à data lida.

```
#include <stdio.h>
#include <string.h>
#include <time.h>
void imp_tempo(struct tm *);
int main(void) {
  struct tm tempo;
  time_t t_c;
  memset(&tempo, 0, sizeof(struct tm));
  printf("dia: ");
  scanf("%d", &(tempo.tm_mday));
  printf("mes: ");
  scanf("%d", &(tempo.tm_mon));
  printf("ano: ");
  scanf("%d", &(tempo.tm_year));
  (tempo.tm_mon)--;
  tempo.tm_year -= 1900;
  imp_tempo(&tempo);
```

continua...

Obtendo data e hora

Exemplo

...continuação

```
  if ((t_c = mktime(&tempo)) ==
                              (time_t) -1) {
    printf("data invalida\n");
  } else {
    imp_tempo(&tempo);
    printf("Equivale a %ld s "
           "desde 1/1/1970\n", t_c);
  }
  return 0;
}
```

continua...

Obtendo data e hora

Exemplo

...continuação

```
void imp_tempo(struct tm *t) {
  char *sem[7] = {"dom", "seg", "ter",
                  "qua", "qui", "sex", "sab"};
  printf("%d h %d min %d s %d/%d/%d ",
              t->tm_hour, t->tm_min, t->tm_sec,
              t->tm_mday, (t->tm_mon + 1),
              (t->tm_year + 1900));
  if ((t->tm_wday < 0) || (t->tm_wday > 6))
    printf("(%d, ", t->tm_wday);
  else
    printf("(%s, ", sem[t->tm_wday]);
  printf("%d dias%s)\n", t->tm_yday,
            t->tm_isdst > 0 ? ", verao" : "");
}
```

Diferença entre datas

```
double difftime(time_t t1, time_t t0)
```

Calcula a diferença entre datas.

Valor de retorno. O valor `t1` – `t0`, expresso em segundos.

Representação textual da data e hora

```
char *asctime(const struct tm *tempo)
```

Converte o tempo armazenado na estrutura apontada por `tempo` em uma cadeia de caracteres contendo contendo a representação dos componentes da estrutura no seguinte formato: `Sun Sep 16 01:03:52 1973\n\0`

Valor de retorno. Um ponteiro para a cadeia de caracteres gerada.

Representação textual da data e hora

```
char *asctime(const struct tm *tempo)
```

Converte o tempo armazenado na estrutura apontada por `tempo` em uma cadeia de caracteres contendo contendo a representação dos componentes da estrutura no seguinte formato: `Sun Sep 16 01:03:52 1973\n\0`

Valor de retorno. Um ponteiro para a cadeia de caracteres gerada.

```
char *ctime(const time_t *tempo)
```

Converte o tempo apontado por `tempo` em uma cadeia de caracteres com um formato idêntico ao da saída produzida por `asctime`.

Valor de retorno. Um ponteiro para a cadeia gerada.

Representação textual da data e hora

```
size_t strftime(char * restrict s, size_t maxqtd,
                    const char * restrict formato,
                const struct tm * restrict tempo)
```

Grava na cadeia apontada por `s` os valores do tempo armazenado na estrutura apontada por `tempo`, segundo o formato definido na cadeia `formato`. As diretivas da cadeia do formato são usadas para converter os valores dos componentes de `tempo` em sua representação textual. O caractere nulo é sempre inserido ao final. No máximo `maxqtd` caracteres são gravados.

Valor de retorno. A quantidade de caracteres gravados, sem contar o caractere nulo ao final, se a conversão foi bem sucedida, ou zero, em caso contrário.

Representação textual da data e hora

Algumas diretivas da função `strftime`:

Dir	Descrição: [Faixa]	Exemplo
`%d`	Dia do mês: [01, 31]	29
`%e`	Dia do mês (os dígitos simples são precedidos de espaço): [1, 31]	29
`%m`	Mês, como um número decimal: [01, 12]	12
`%Y`	Ano.	2014
`%C`	Ano dividido por 100 e truncado para um valor inteiro.	20
`%y`	Ano, dois últimos dígitos: [00, 99]	14
`%D`	Mês, dia e ano (equivalente a `%m/%d/%y`).	12/29/14
`%u`	Dia da semana, como um número decimal (segunda como o dia 1): [1, 7]	1
`%A`	Nome do dia da semana por extenso.	segunda
`%B`	Nome do mês por extenso.	dezembro
`%H`	Hora (24 horas): [00, 23]	17
`%I`	Hora (12 horas): [01, 12]	05
`%M`	Minuto: [00, 59]	03
`%S`	Segundo: [00, 60]	28
`%X`	Hora na localização em vigor.	17:03:28

Tempo de processador

Para um dado processo o tempo de processador é dividido em:

Tempo do processo. Tempo de uso efetivo da CPU para executar as operações do próprio processo.

Tempo do sistema. Tempo de uso efetivo da CPU para executar as operações do sistema solicitadas pelo próprio processo.

Tempo dos processos filhos. Tempo de uso efetivo da CPU para executar as operações dos processos filhos.

Tempo do sistema devotado aos processos filhos. Tempo de uso efetivo da CPU para executar as operações do sistema solicitadas pelos processos filhos.

Tempo de processador

Para um dado processo o tempo de processador é dividido em:

Tempo do processo. Tempo de uso efetivo da CPU para executar as operações do próprio processo.

Tempo do sistema. Tempo de uso efetivo da CPU para executar as operações do sistema solicitadas pelo próprio processo.

Tempo dos processos filhos. Tempo de uso efetivo da CPU para executar as operações dos processos filhos.

Tempo do sistema devotado aos processos filhos. Tempo de uso efetivo da CPU para executar as operações do sistema solicitadas pelos processos filhos.

- O tipo `clock_t` é um tipo aritmético usado para representar o tempo de processador.
- A macro `CLOCKS_PER_SEC` representa a quantidade de pulsos de relógio em um segundo.

Tempo de processador

```
clock_t clock(void)
```

Calcula o tempo de uso efetivo da CPU pelo processo desde o início de um evento específico, relacionado com a ativação do próprio processo. O evento a partir do qual o tempo é calculado é dependente da implementação.

Valor de retorno. Tempo de uso da CPU, como uma quantidade de pulsos de relógio do processador, ou -1, se este tempo não está disponível ou se o valor não pode ser representado no tipo `clock_t`.

Para se obter o tempo de CPU em segundos deve-se dividir o resultado da função `clock` pela macro `CLOCKS_PER_SEC`.

Tempo de processador

Exemplo

O programa ao lado calcula o valor aproximado de π usando a série

$$\frac{1}{1^3} + \frac{1}{3^3} + \frac{1}{5^3} \cdots ,$$

com uma quantidade de termos informada pelo usuário. Ao final, o programa imprime, além do valor calculado, os tempos de sala e CPU dispendidos em sua execução.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int main(void) {
  time_t tsala_ini, tsala_fim;
  double tproc_ini, tproc_fim,
         tproc, val = 0.0;
  int n;
  tsala_ini = time(NULL);
  tproc_ini = (double)clock();
  printf("Qtd. termos da serie: ");
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    val = val + 1.0/pow(2.0 * i + 1.0, 3.0);
  }
```

continua...

Tempo de processador

Exemplo

...continuação

```
  printf("valor da serie = %f\n", val);
  tsala_fim = time(NULL);
  tproc_fim = (double)clock();
  tproc = (tproc_fim - tproc_ini)/
          CLOCKS_PER_SEC;
  printf("tempo sala: %f s\n",
          difftime(tsala_fim, tsala_ini));
  printf("processamento: %f s\n", tproc);
  return 0;
}
```

Temporizadores

- Um temporizador é um mecanismo que dispara após um certo intervalo de tempo, provocando alguma ação em função desse disparo.
- Os temporizadores são normalmente implementados usando-se
  - a função `difftime` ou
  - as funções `sleep` e `alarm` do cabeçalho `unistd.h` (parte do padrão POSIX).

Temporizadores

```
unsigned int sleep(unsigned int seg)
```

Suspende o processamento, que é reiniciado aproximadamente `seg` segundos após a execução da função, ou após o lançamento de algum sinal de interrupção para o programa.

Valor de retorno. O valor 0, se o retorno ocorre em função do transcurso dos segundos indicados pelo argumento, ou a quantidade de segundos remanescentes, se o retorno ocorre antes de `seg` segundos, em função de algum sinal de interrupção.

Temporizadores

`unsigned int alarm(unsigned int seg)`

Ajusta o alarme do relógio de tempo real para disparar após transcorridos `seg` segundos. Por ocasião do disparo, o sinal `SIGALRM` é lançado e pode ser capturado por uma função de tratamento de sinais.

Valor de retorno. A quantidade de segundos que falta para o disparo anterior do alarme, ou o valor 0, se não houver ajuste anterior para o alarme.

Temporizadores

Exemplo

O programa ao lado aciona o alarme para disparar após a quantidade de segundos informada pelo usuário.

O sinal SIGALRM é capturado pela função trata_alarme.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
volatile sig_atomic_t continua = 1;
void trata_alarme(int);
int main(void) {
  unsigned int seg, qtd = 0, num;
  signal(SIGALRM, trata_alarme);
  printf("Segundos p/interromper: ");
  scanf("%u", &seg);
  alarm(seg);
```

continua...

Temporizadores

Exemplo

...continuação

```
  printf("Digite inteiros > 1.000\n");
  do {
    scanf("%*[^\n]"); scanf("%*c");
    printf("num: ");
    if ((scanf("%u", &num) == 1) &&
        (num > 1000U)) {
      qtd++;
    }
  } while (continua == 1);
  printf("\nDigitou %u nums em %u s.\n",
                                 qtd, seg);
  return 0;
}
void trata_alarme(int s) {
  continua = 0;
  signal(s, trata_alarme);
}
```

Biblioteca matemática

- As funções matemáticas são declaradas no cabeçalho `math.h`.
- Para cada função existem versões para os tipos `double`, `float` e `long double`
- Deve-se verificar na descrição das funções os tipos de erro que podem ocorrer na execução de cada função, e como eles são indicados.
- Em alguns ambientes de compilação a biblioteca que as implementa, `libm.a` ou `libm.so`, não é automaticamente incorporada, sendo necessário fazer referência explícita a ela:

```
gcc prog -c prog.c -lm
```

Constantes numéricas

- O padrão ISO/IEC 9989:1999 não especifica constantes matemáticas usuais, como π e **e**.
- Muitas implementações da linguagem fornecem essas constantes como uma extensão.
  - O compilador gcc define as seguintes constantes, como macros, no arquivo-cabeçalho `math.h`, que podem ser usadas quando as macros `_GNU_SOURCE` ou `_XOPEN_SOURCE` estão definidas:

Macro	Valor	Macro	Valor	Macro	Valor
`M_E`	$\mathbf{e}$	`M_PI`	π	`M_2_SQRTPI`	$2/\sqrt{\pi}$
`M_LOG2E`	$\log_2 \mathbf{e}$	`M_PI_2`	$\pi/2$	`M_SQRT2`	$\sqrt{2}$
`M_LOG10E`	$\log_{10} \mathbf{e}$	`M_PI_4`	$\pi/4$	`M_SQRT1_2`	$1/\sqrt{2}$
`M_LN2`	$\log_{\mathbf{e}} 2$	`M_1_PI`	$1/\pi$		
`M_LN10`	$\log_{\mathbf{e}} 10$	`M_2_PI`	$2/\pi$		

Constantes numéricas

Para aumentar a portabilidade aconselha-se o cálculo das constantes numéricas:

```
const double PI = 2 * atan(1.0/0.0);
const double E = exp(1.0);
```

Números randômicos

`int rand(void)`

Gera uma sequência de números inteiros pseudorrandômicos na faixa [0, `RAND_MAX`]. A cada chamada da função um novo número da sequência é obtido.

Valor de retorno. Um número da sequência de números pseudorrandômicos na faixa [0, `RAND_MAX`].

A macro `RAND_MAX` é definida no cabeçalho `stdlib.h`, sendo no mínimo igual a 32.767.

Números randômicos

```
void srand(unsigned int semente)
```

Define o valor `semente` como sendo a semente para a próxima sequência de números pseudorrandômicos gerada pela função `rand`.

Valor de retorno. Não tem.

Números randômicos

Exemplo

O programa ao lado imprime duas sequências de 100 números randômicos entre 20 e 42, inclusive.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void) {
  unsigned int raiz = 234u;
  int ini = 20, fim = 42;
  srand(raiz);
  for (int i = 0; i < 100; i++) {
    printf("%d ",
           ini + rand() % (fim - ini + 1));
  }
  printf("\n\nSegunda seq randomica:\n");
  srand((unsigned int)time(NULL));
  for (int i = 0; i < 100; i++) {
    printf("%d ",
           ini + rand() % (fim - ini + 1));
  }
  return 0;
}
```

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc

Elementos de programação em C

Diretivas de pré-processamento

Francisco A. C. Pinheiro, *Elementos de Programação em C*, Bookman, 2012.

Visite os sítios do livro para obter material adicional: www.bookman.com.br e www.facp.pro.br/livroc

Sumário

Inclusão de arquivos

`#include <arqsis>`	Inclui o arquivo do sistema `arqsis`.
`#include "arqusu"`	Inclui o arquivo do usuário `arqusu`.

Inclusão de arquivos

`#include <arqsis>` Inclui o arquivo do sistema `arqsis`.

`#include "arqusu"` Inclui o arquivo do usuário `arqusu`.

As seguintes opções modificam a ordem de pesquisa dos diretórios na busca por arquivos durante a compilação:

`-I`*dir* Coloca *dir* no início da lista dos diretórios que são pesquisados na busca dos arquivos identificados pelas diretivas `#include`.

`-iquote`*dir* Coloca *dir* no início da lista dos diretórios que são pesquisados na busca dos arquivos do usuário, se esses arquivos não estiverem no diretório corrente.

Inclusão de arquivos — ordem de pesquisa

Com as opções `-I` e `-iquote`, a seguinte ordem de pesquisa é adotada:

- Os arquivos do sistema são pesquisados primeiro nos diretórios especificados pelas opções `-I`, e por último nos diretórios padrões do sistema.
- Os arquivos do usuário são pesquisados primeiro no diretório corrente, a seguir nos diretórios especificados pelas opções `iquote`, depois nos diretórios especificados pelas opções `-I`, e por último nos diretórios padrões do sistema.

Definindo constantes

`#define` ⟨*nome_macro*⟩ ⟨*expressão_definidora*⟩

- Durante o pré-processamento todas as referências a macros são substituídas por sua definição.
- A expressão definidora de uma macro não é avaliada no processo de substituição, que é puramente textual.
- A expressão resultante da substituição é avaliada durante a compilação.
- As macros podem ser usadas na definição de outras macros:
  - Após cada substituição o texto resultante é reanalisado e, se nele houver macros, novas substituições têm efeito.
  - Entretanto, se em algum momento o texto resultante contém o nome da macro sendo substituída, esse nome não é mais objeto de avaliação.

Definindo constantes

Exemplo

A sequência de definições ao lado faz com que VF seja substituída por:

```
#define VP 500
#define ANOS 13
#define MESES 1
#define JUROS_PERC 0.6
#define JUROS JUROS_PERC / 100
#define PERIODO ANOS * 12 + MESES
#define VF VP * pow((1 + JUROS), PERIODO)
```

```
500 * pow((1 + 0.6 / 100), 13 * 12 + 1)
```

Definindo constantes

Exemplo

A sequência de definições ao lado faz com que VF seja substituída por:

```
#define VP 500
#define ANOS 13
#define MESES 1
#define JUROS_PERC 0.6
#define JUROS JUROS_PERC / 100
#define PERIODO ANOS * 12 + MESES
#define VF VP * pow((1 + JUROS), PERIODO)
```

```
500 * pow((1 + 0.6 / 100), 13 * 12 + 1)
```

```
VF → VP * pow((1 + JUROS), PERIODO)
```

Definindo constantes

Exemplo

A sequência de definições ao lado faz com que VF seja substituída por:

```
#define VP 500
#define ANOS 13
#define MESES 1
#define JUROS_PERC 0.6
#define JUROS JUROS_PERC / 100
#define PERIODO ANOS * 12 + MESES
#define VF VP * pow((1 + JUROS), PERIODO)
```

```
500 * pow((1 + 0.6 / 100), 13 * 12 + 1)
```

```
VF → VP * pow((1 + JUROS), PERIODO)
   → 500 * pow((1 + JUROS), PERIODO)
```

Definindo constantes

Exemplo

A sequência de definições ao lado faz com que VF seja substituída por:

```
#define VP 500
#define ANOS 13
#define MESES 1
#define JUROS_PERC 0.6
#define JUROS JUROS_PERC / 100
#define PERIODO ANOS * 12 + MESES
#define VF VP * pow((1 + JUROS), PERIODO)
```

```
500 * pow((1 + 0.6 / 100), 13 * 12 + 1)
```

```
VF → VP * pow((1 + JUROS), PERIODO)
   → 500 * pow((1 + JUROS), PERIODO)
   → 500 * pow((1 + JUROS_PERC / 100), PERIODO)
```

Definindo constantes

Exemplo

A sequência de definições ao lado faz com que VF seja substituída por:

```
#define VP 500
#define ANOS 13
#define MESES 1
#define JUROS_PERC 0.6
#define JUROS JUROS_PERC / 100
#define PERIODO ANOS * 12 + MESES
#define VF VP * pow((1 + JUROS), PERIODO)
```

```
500 * pow((1 + 0.6 / 100), 13 * 12 + 1)
```

```
VF → VP * pow((1 + JUROS), PERIODO)
   → 500 * pow((1 + JUROS), PERIODO)
   → 500 * pow((1 + JUROS_PERC / 100), PERIODO)
```

e assim por diante.

Simulando funções

`#define` ⟨*nome_macro*⟩(⟨*lista_parâmetros*⟩) ⟨*expressão_definidora*⟩

- A lista de parâmetros deve vir imediatamente após o nome da macro, sem espaço entre eles.
- Os parâmetros são especificados entre parênteses, separados por vírgula.
- Os parâmetros de uma macro podem não ser usados na expressão definidora.
- As referências a uma macro parametrizada devem respeitar o número de parâmetros de sua definição.

Simulando funções

```
#define f(x) 2 * (x)
```

```
#define abs(val, x)
                  if ((x) < 0) (val) = -(x); else (val) = (x)
```

```
#define g(a, xy)(a) > 0?(a) : (xy)
```

```
#define fun(a, b, c) 2 * (a) + (b)
```

Simulando funções

```
#define f(x) 2 * (x)
f(x + 3) → 2 * (x + 3)
```

```
#define abs(val, x)
                 if ((x) < 0) (val) = -(x); else (val) = (x)
```

#define g(a, xy)$(a) > 0?(a) : (xy)$

```
#define fun(a, b, c) 2 * (a) + (b)
```

Simulando funções

```
#define f(x) 2 * (x)
f(x + 3) → 2 * (x + 3)
```

```
#define abs(val, x)
                 if ((x) < 0) (val) = -(x); else (val) = (x)
abs(&num, -2) →
             if ((-2) < 0) (&num) = -(-2); else (&num) = (-2)
```

#define g(a, xy)$(a) > 0?(a) : (xy)$

```
#define fun(a, b, c) 2 * (a) + (b)
```

Simulando funções

```
#define f(x) 2 * (x)
f(x + 3) → 2 * (x + 3)
```

```
#define abs(val, x)
                    if ((x) < 0) (val) = -(x); else (val) = (x)
abs(&num, -2) →
              if ((-2) < 0) (&num) = -(-2); else (&num) = (-2)
```

#define g(a, xy)$(a) > 0?(a) : (xy)$

g(x + y, 21) → (x + y) > 0 ? (x + y) : (21).

```
#define fun(a, b, c) 2 * (a) + (b)
```

Simulando funções

```
#define f(x) 2 * (x)
f(x + 3) → 2 * (x + 3)
```

```
#define abs(val, x)
                  if ((x) < 0) (val) = -(x); else (val) = (x)
abs(&num, -2) →
              if ((-2) < 0) (&num) = -(-2); else (&num) = (-2)
```

#define g(a, xy)$(a) > 0?(a) : (xy)$

g(x + y, 21) → (x + y) > 0 ? (x + y) : (21).

```
#define fun(a, b, c) 2 * (a) + (b)
fun(2 * c, c, y) → 2 * (2 + c) + (c).
```

Simulando funções — argumentos variáveis

- Os parâmetros variáveis são identificados por reticências na lista de parâmetros.
- Na referência, os argumentos que correspondem aos parâmetros variáveis (incluindo as vírgulas que os separam) são tratados como um único valor, substituindo o identificador `__VA_ARGS__`.

Simulando funções — argumentos variáveis

Exemplo

A macro

```
#define soma(x, ...)  x + adiciona(__VA_ARGS__)
```

resulta nas seguintes substituições:

```
soma(2, 3)
soma(a, b, c)
soma(6, 4, x + 3, c)
```

Simulando funções — argumentos variáveis

Exemplo

A macro

```
#define soma(x, ...)  x + adiciona(__VA_ARGS__)
```

resulta nas seguintes substituições:

```
soma(2, 3)          →  2 + adiciona(3)
soma(a, b, c)
soma(6, 4, x + 3, c)
```

Simulando funções — argumentos variáveis

Exemplo

A macro

```
#define soma(x, ...)  x + adiciona(__VA_ARGS__)
```

resulta nas seguintes substituições:

```
soma(2, 3)            →  2 + adiciona(3)
soma(a, b, c)         →  a + adiciona(b, c)
soma(6, 4, x + 3, c)
```

Simulando funções — argumentos variáveis

Exemplo

A macro

```
#define soma(x, ...)  x + adiciona(__VA_ARGS__)
```

resulta nas seguintes substituições:

```
soma(2, 3)           →  2 + adiciona(3)
soma(a, b, c)        →  a + adiciona(b, c)
soma(6, 4, x + 3, c) →  6 + adiciona(4, x + 3, c)
```

Representando argumentos como cadeias de caracteres

- O símbolo # é tratado como um operador de substituição literal, quando antecede o identificador de um parâmetro na definição de uma macro.
- Seu efeito é delimitar com aspas a expressão usada como argumento na referência à macro:
  - a expressão $\langle arg \rangle$ é transformada em "$\langle arg \rangle$".

Representando argumentos como cadeias de caracteres

Exemplo

A macro

```
#define imprime(x) printf("arg " #x " = %d\n", (x))
```

resulta nas seguintes substituições:

```
imprime(x)
```

```
imprime(4 + a)
```

Representando argumentos como cadeias de caracteres

Exemplo

A macro

```
#define imprime(x) printf("arg " #x " = %d\n", (x))
```

resulta nas seguintes substituições:

`imprime(x)`	$\longrightarrow$	`printf("arg ""x"" = %d\n", (x))`
`imprime(4 + a)`		

Representando argumentos como cadeias de caracteres

Exemplo

A macro

```
#define imprime(x) printf("arg " #x " = %d\n", (x))
```

resulta nas seguintes substituições:

```
imprime(x)       ⟶  printf("arg ""x"" = %d\n", (x))
                 ⟶  printf("arg x = %d\n", (x))
```

```
imprime(4 + a)
```

Representando argumentos como cadeias de caracteres

Exemplo

A macro

```
#define imprime(x) printf("arg " #x " = %d\n", (x))
```

resulta nas seguintes substituições:

```
imprime(x)       →  printf("arg ""x"" = %d\n", (x))
                 →  printf("arg x = %d\n", (x))
imprime(4 + a)   →  printf("arg ""4 + a"" = %d\n", (4 + a))
```

Representando argumentos como cadeias de caracteres

Exemplo

A macro

```
#define imprime(x) printf("arg " #x " = %d\n", (x))
```

resulta nas seguintes substituições:

`imprime(x)`	$\longrightarrow$	`printf("arg ""x"" = %d\n", (x))`
	$\longrightarrow$	`printf("arg x = %d\n", (x))`
`imprime(4 + a)`	$\longrightarrow$	`printf("arg ""4 + a"" = %d\n", (4 + a))`
	$\longrightarrow$	`printf("arg 4 + a = %d\n", (4 + a))`

Concatenando argumentos

O operador `##` na definição de uma macro causa a concatenação do termo que o precede com o termo que o segue imediatamente:

- a expressão `a ## b` produz `ab`.

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w
```

```
#define imp(x,y,z)
        printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);
arg(1,2, ,869);
arg( , , ,869);

imp(1, 2, 3);
imp(1,2,);
imp( , , );
```

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w

#define imp(x,y,z)
        printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);  →  int val123 = 869;
arg(1,2, ,869);
arg( , , ,869);

imp(1, 2, 3);
imp(1,2,);
imp( , , );
```

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w

#define imp(x,y,z)
        printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);   →   int val123 = 869;
arg(1,2, ,869);      →   int val12 = 869;
arg( , , ,869);

imp(1, 2, 3);
imp(1,2,);
imp( , , );
```

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w

#define imp(x,y,z)
        printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);   →   int val123 = 869;
arg(1,2, ,869);      →   int val12 = 869;
arg( , , ,869);      →   int val = 869;

imp(1, 2, 3);
imp(1,2,);
imp( , , );
```

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w

#define imp(x,y,z)
        printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);   →  int val123 = 869;
arg(1,2, ,869);      →  int val12 = 869;
arg( , , ,869);      →  int val = 869;

imp(1, 2, 3);        →  printf("val123 = %d\n", val123);
imp(1,2,);
imp( , , );
```

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w

#define imp(x,y,z)
        printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);   →  int val123 = 869;
arg(1,2, ,869);      →  int val12 = 869;
arg( , , ,869);      →  int val = 869;

imp(1, 2, 3);        →  printf("val123 = %d\n", val123);
imp(1,2,);           →  printf("val12 = %d\n", val12);
imp( , , );
```

Concatenando argumentos

Exemplo

As macros

```
#define arg(x, y, z, w) int val ## x ## y ## z = w

#define imp(x,y,z)
       printf("val" #x #y #z " = %d\n", val ## x ## y ## z)
```

produzem as seguintes substituições:

```
arg(1, 2, 3, 869);   →  int val123 = 869;
arg(1,2, ,869);      →  int val12 = 869;
arg( , , ,869);      →  int val = 869;

imp(1, 2, 3);        →  printf("val123 = %d\n", val123);
imp(1,2,);           →  printf("val12 = %d\n", val12);
imp( , , );          →  printf("val = %d\n", val);
```

Simulando funções — recomendações

Os parâmetros devem ser delimitados por parênteses

```
#define mult(x, y) (x) * (y)
```

é preferível a

```
#define mult(x, y) x * y
```

Simulando funções — recomendações

Os parâmetros devem ser delimitados por parênteses

```
#define mult(x, y) (x) * (y)
mult(x + 3, x + 3) → (x + 3) * (x + 3)
```

é preferível a

```
#define mult(x, y) x * y
mult(x + 3, x + 3) → x + 3 * x + 3
```

Simulando funções — recomendações

Deve-se evitar argumentos com efeitos colaterais

```
#define dobro(x) (x) + (x)
```

é diferente de

```
int dobro(int x) {return x + x;}
```

Simulando funções — recomendações

Deve-se evitar argumentos com efeitos colaterais

```
#define dobro(x) (x) + (x)
```

```
x = 4;
y = dobro(++x);
```

resulta em

```
x = 6
y = 12
```

é diferente de

```
int dobro(int x) {return x + x;}
```

```
x = 4;
y = dobro(++x);
```

resulta em

```
x = 5
y = 10
```

Tornando definições sem efeito

- `#define LINUX` define a macro `LINUX`
- `#undef WINDOWS` torna sem efeito a definição da macro `LINUX`
- Em tempo de compilação pode-se usar as opções `-D` e `-U` para definir e tornar as definições sem efeito:
  - `gcc -o prog prog.c -std=c99 -DLINUX -UWINDOWS`

Testando a definição de macros

Operador `defined`

A expressão `defined NOME` ou `defined(NOME)` resulta

- no valor 1 (verdadeiro), se a macro `NOME` está definida,
- ou no valor 0 (falso), em caso contrário.

Operador `!defined`

A expressão `!defined NOME` ou `!defined(NOME)` resulta

- no valor 1 (verdadeiro), se a macro `NOME` não está definida, ou
- no valor 0 (falso), em caso contrário.

Inclusão condicional de código

O comando `#if` e suas variações são usados para inclusão condicional de código:

```
#if defined LINUX
  <texto-1>
#endif
<texto-2>
```

```
#if !defined LINUX
  <texto-1>
#else
  <texto-2>
#endif
<texto-3>
```

```
#if !defined LINUX
  <texto-1>
#else
  <texto-2>
  #if defined MAC
    <texto-3>
  #endif
#endif
<texto-4>
```

Inclusão condicional de código

`#ifdef`	é abreviação de	`#if defined`
`#ifndef`	é abreviação de	`#if !defined`
`#elif`	é abreviação de	`#else #if`
`#endif`	termina os comandos	`#if` e `#elif`
`#endelif`	termina o comando	`#elif`

```
#ifdef UNIX
  <texto-1>
#elif defined WINDOWS
  <texto-2>
#endelif
#endif
<texto-3>
```

```
#ifdef UNIX
  <texto-1>
#elif defined WINDOWS
  <texto-2>
#endif
<texto-3>
```

```
#ifndef SUN
  <texto-1>
#elif defined PC
  <texto-2>
#elif defined MAC
  <texto-3>
#endif
<texto-4>
```

Avaliação da condição

1. Os identificadores que são nomes de macros definidas, e não são operandos do operador `defined`, são substituídos por suas definições.
2. As operações `defined` são avaliadas, resultando
   - no valor 1, se a macro usada como operando está definida, ou
   - no valor 0, em caso contrário.
3. Os identificadores remanescentes são substituídos por 0.
4. A expressão resultante é avaliada.
   - A expressão resultante deve ser uma expressão inteira constante, mas não pode conter o operador `sizeof` nem o operador de conversão de tipo.

Avaliação da condição

Exemplo

Para as macros ao lado, a tabela a seguir mostra como a condição da diretiva #if é avaliada.

```
#define E1
#undef E2
#define E2 12
#define E3
#undef E3
#define E5 LINUX
#define SUNOS  1
```

Antes	Após subst.	Aval	
`#if defined E1`	`#if defined E1`	`#if 1`	E1 está definida.
`#if !defined E2`	`#if !defined E2`	`#if 0`	E2 está definida.
`#if !defined E3`	`#if !defined E3`	`#if 1`	E3 não está definida.
`#if E2 == 3 * (2<<1)`	`#if 12 == 3 * (2<<1)`	`#if 1`	E2 é substituída por 12.
`#if (E5 == SUNOS)`	`#if (0 == 1)`	`#if 0`	E5 é substituída por LINUX, LINUX é substituída por 0, SUNOS é substituída por 1.

Algumas macros predefinida

Macro	Definição
`__DATE__`	Data do pré-processamento da unidade de compilação
`__TIME__`	Hora do pré-processamento da unidade de compilação
`__FILE__`	Nome do arquivo em que é utilizada.
`__func__`	Nome da função em que é utilizada (não é, de fato, uma macro).
`__FUNCTION__`	Nome da função em que é utilizada (extensão do gcc).
`__LINE__`	Número da linha do arquivo-fonte na qual ocorre a referência à macro.

Algumas macros predefinida

Exemplo

```
    printf("erro: arquivo %s, linha %d.\n",
            arq, linha);
    /* codigo omitido */
  }
```

A função `mostra_erro` pode ser chamada com o nome do arquivo e a linha em que ocorreu o erro:

`mostra_erro(__FILE__, __LINE__)`

Erros de pré-processamento

Exemplo

No trecho de programa a seguir, se a macro QTD for menor ou igual a 20, a compilação é interrompida na fase de pré-processamento produzindo uma mensagem de erro que inclui o texto "tamanho QTD invalido".

```
#if QTD > 20
  int vet[QTD];
#else
  #error tamanho QTD invalido
#endif
```

Bibliografia

ISO/IEC
C Programming Language Standard
ISO/IEC 9899:2011, International Organization for Standardization; International Electrotechnical Commission, 3rd edition, WG14/N1570 Committee final draft, abril de 2011.

Francisco A. C. Pinheiro
Elementos de programação em C
Bookman, Porto Alegre, 2012.
www.bookman.com.br, www.facp.pro.br/livroc